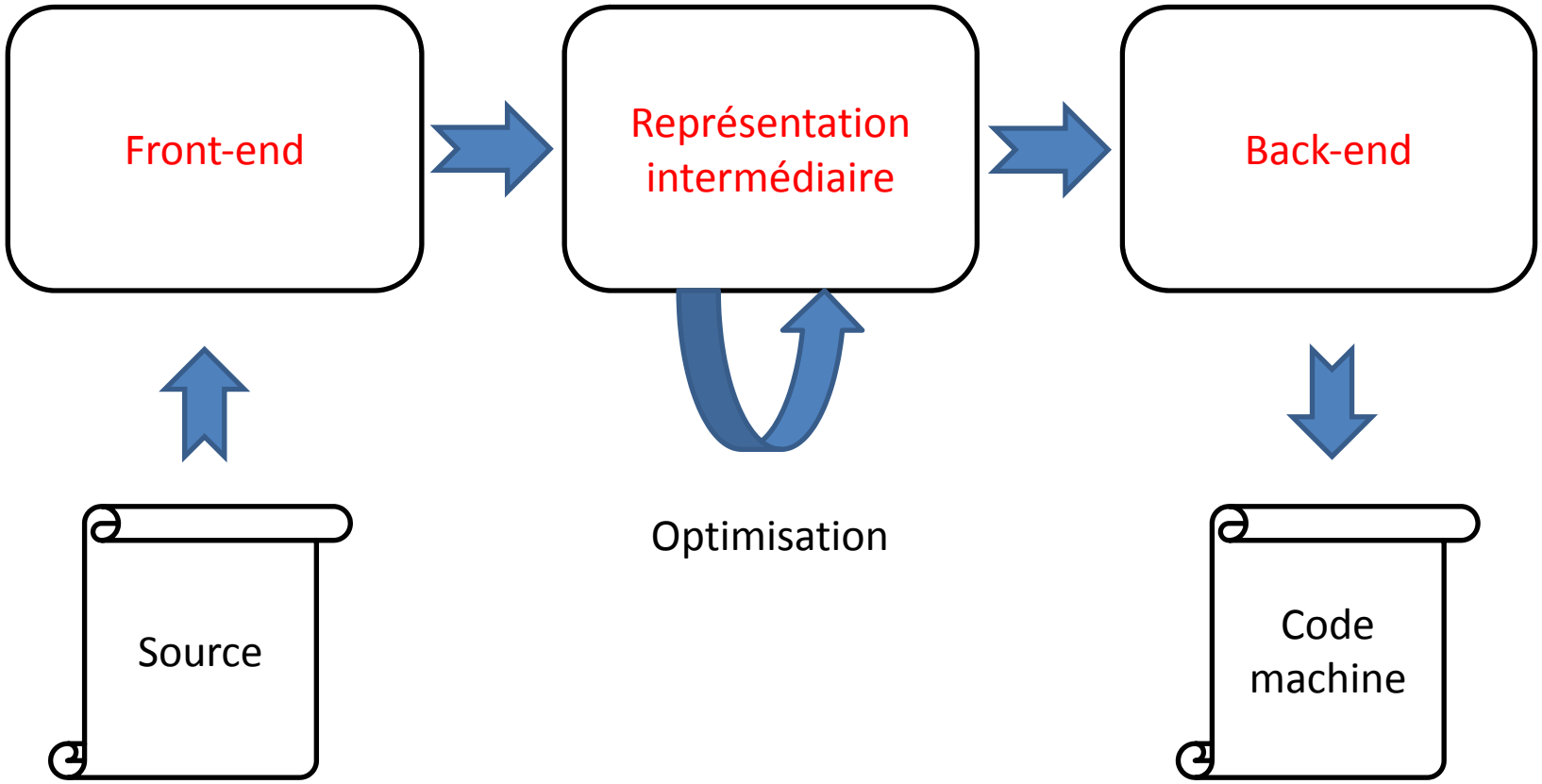
Génération de code machine

Christophe Alias

Contexte


Front-end
Représentation intermédiaire
Back-end
Source
Optimisation
Code machine

Objectifs

- Produire du code machine correct et efficace
- Correct: le code machine est *équivalent* au programme
- Efficace: “bonne” utilisation des ressources (instructions, registres, caches)

Méthode directe

Entrée: CFG avec pseudo-code

- Allouer un slot de pile par temporaire
- Traduire directement le pseudo-code:

```
[[t = t1 + t2]] =
  mov rax,[rbp-slot(t1)]
  mov rbx,[rbp-slot(t2)]
  add rax,rbx
  mov [rbp-slot(t)],rax
```

Méthode classique

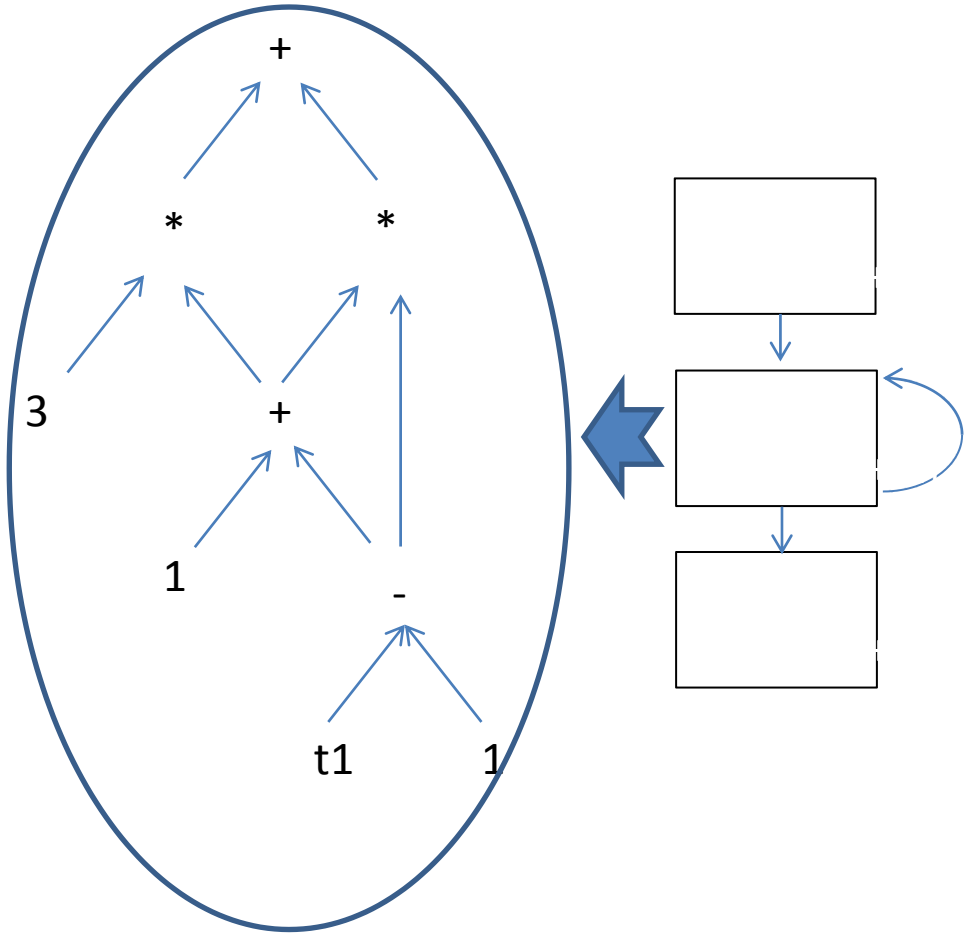
Entrée: CFG avec DAG

Pour chaque DAG:

- Sélection des instructions
- Ordonnancement

Puis:

- Allocation des registres



Méthode classique

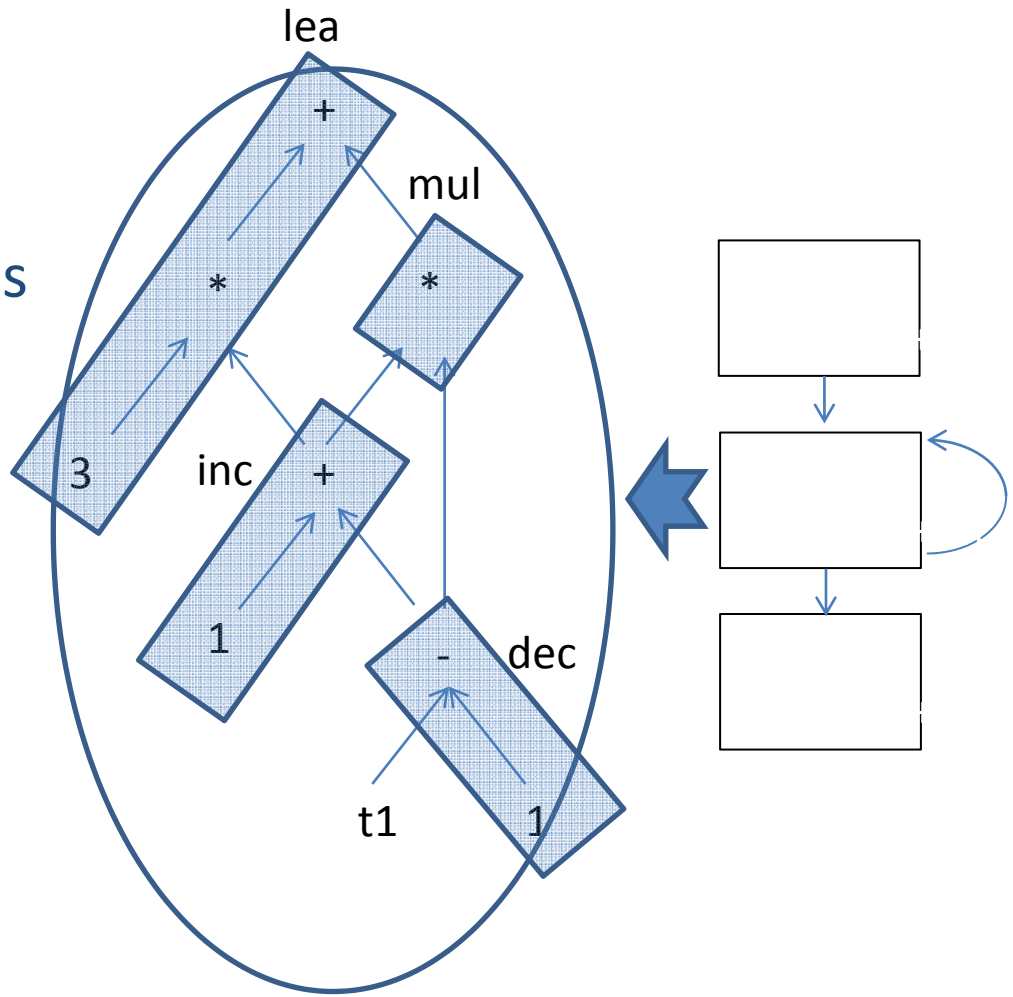
Entrée: CFG avec DAG

Pour chaque DAG:

- Sélection des instructions
- Ordonnancement

Puis:

- Allocation des registres



Méthode classique

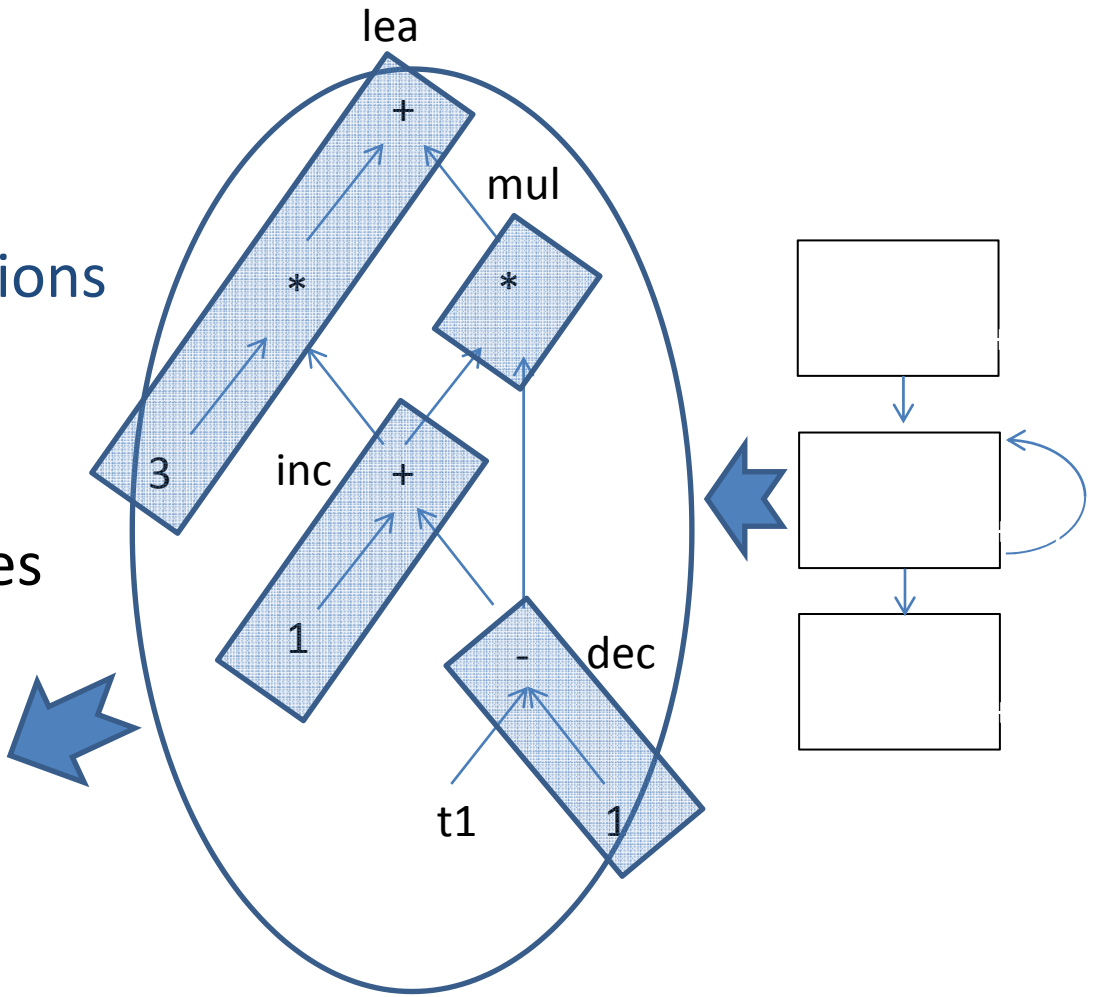
Entrée: CFG avec DAG

Pour chaque DAG:

- Sélection des instructions
- Ordonnancement

Puis:

- Allocation des registres




```
dec t1
mov t2,t1
mov t3,t1
inc t2
mov t4,t2
mov t5,t2
…
```

Méthode classique

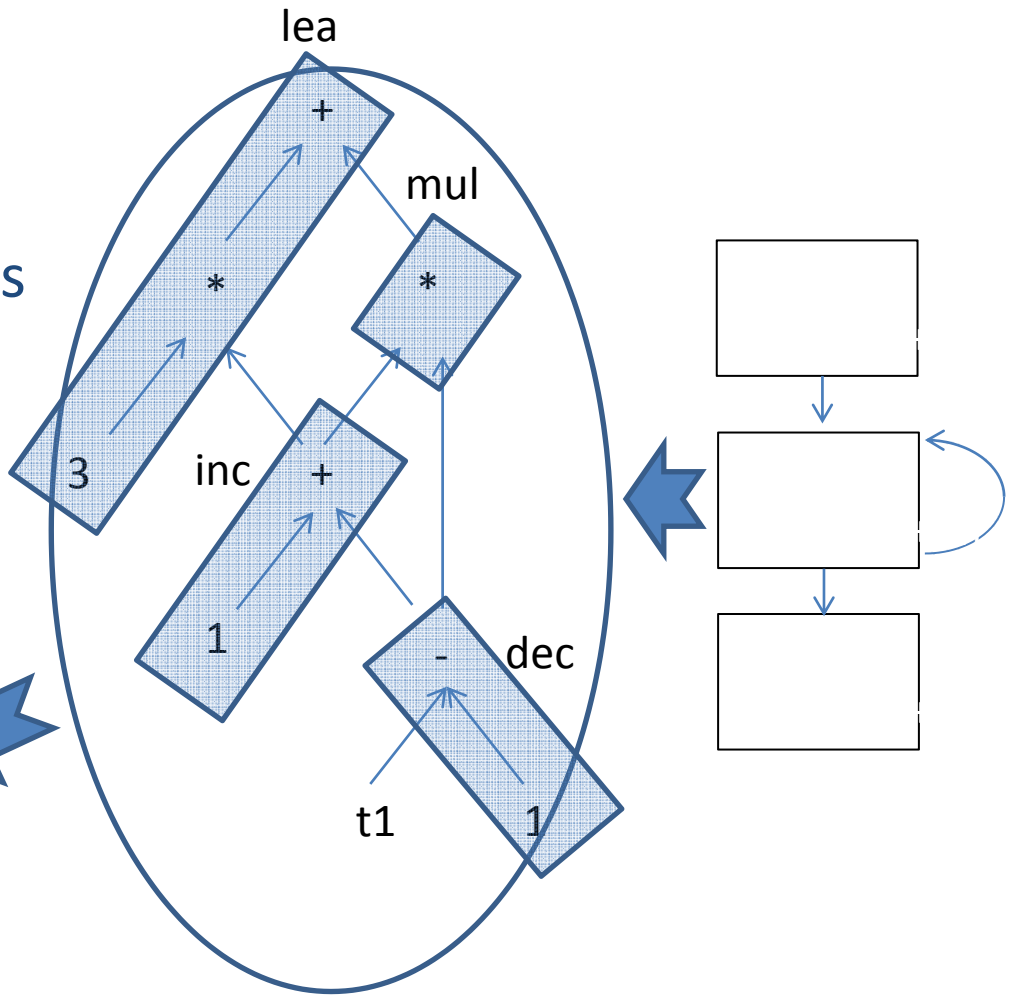
Entrée: CFG avec DAG

Pour chaque DAG:

- Sélection des instructions
- Ordonnancement

Puis:

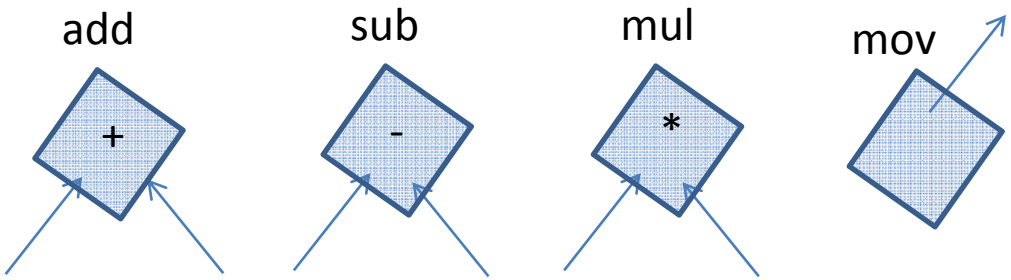
- Allocation des registres

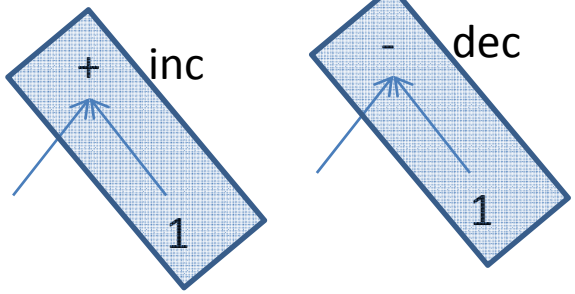


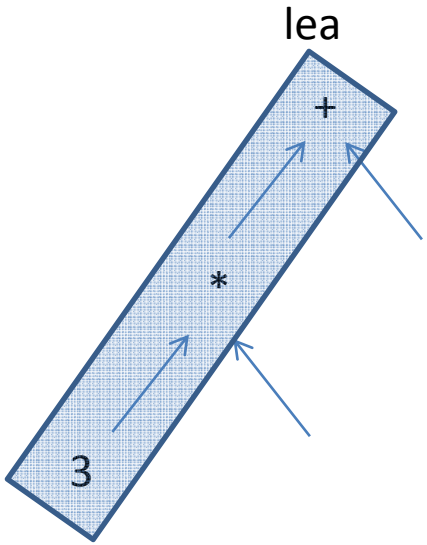

```
dec rax
mov rbx,rax
inc rax
mov rcx,rax
…
```

Sélection des instructions

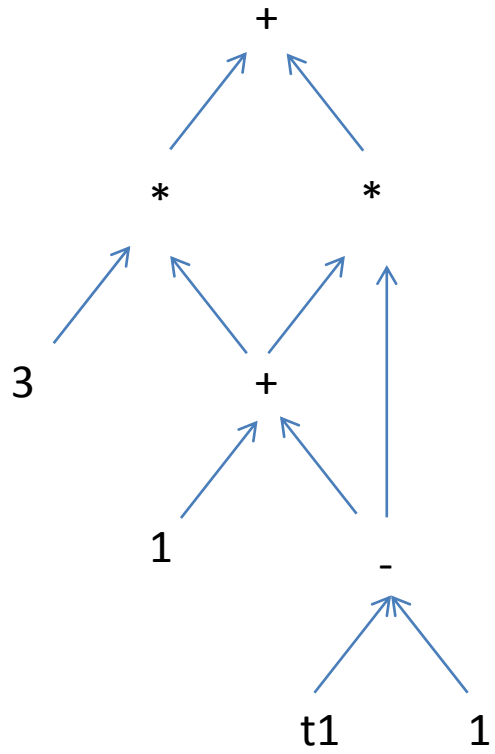
Trouver un pavage de coût minimum





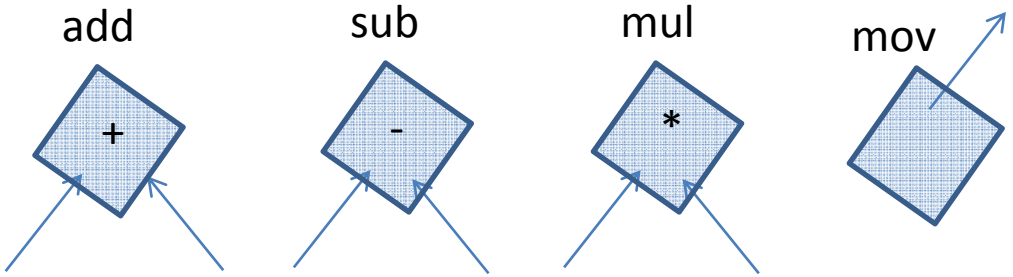


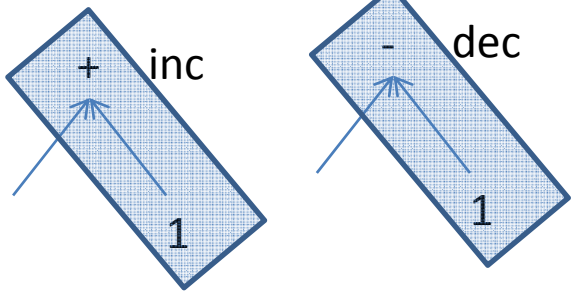
Et permutations…

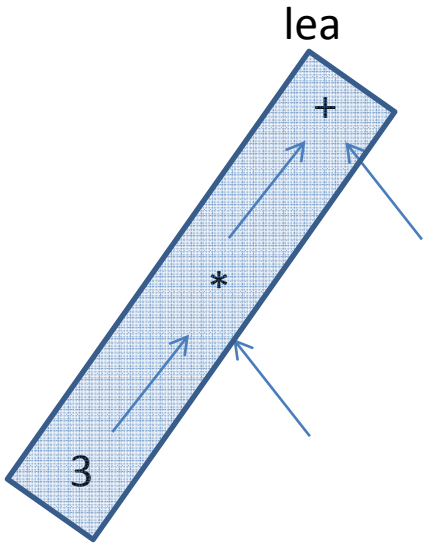


Sélection des instructions

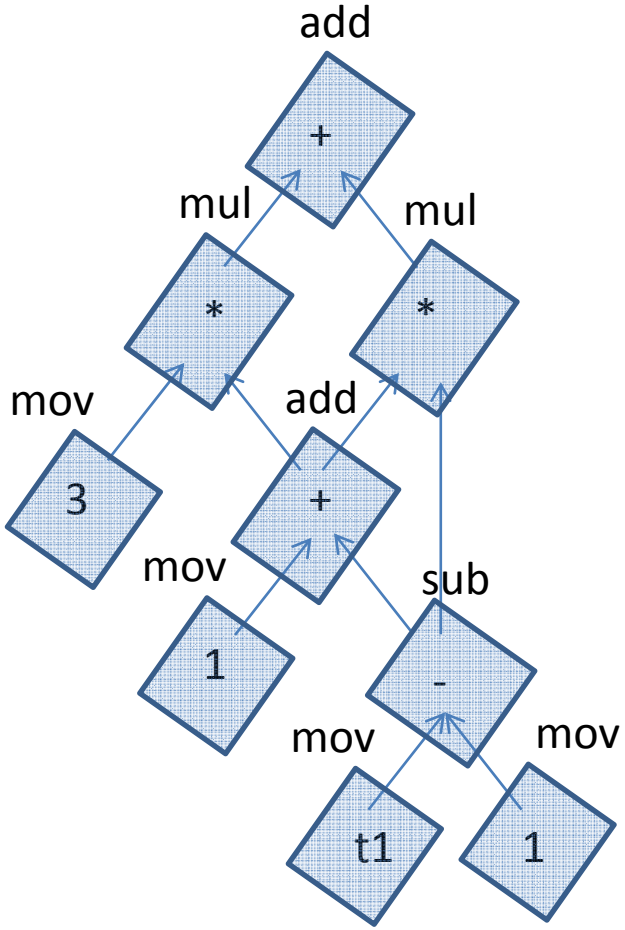
Trouver un pavage de coût minimum





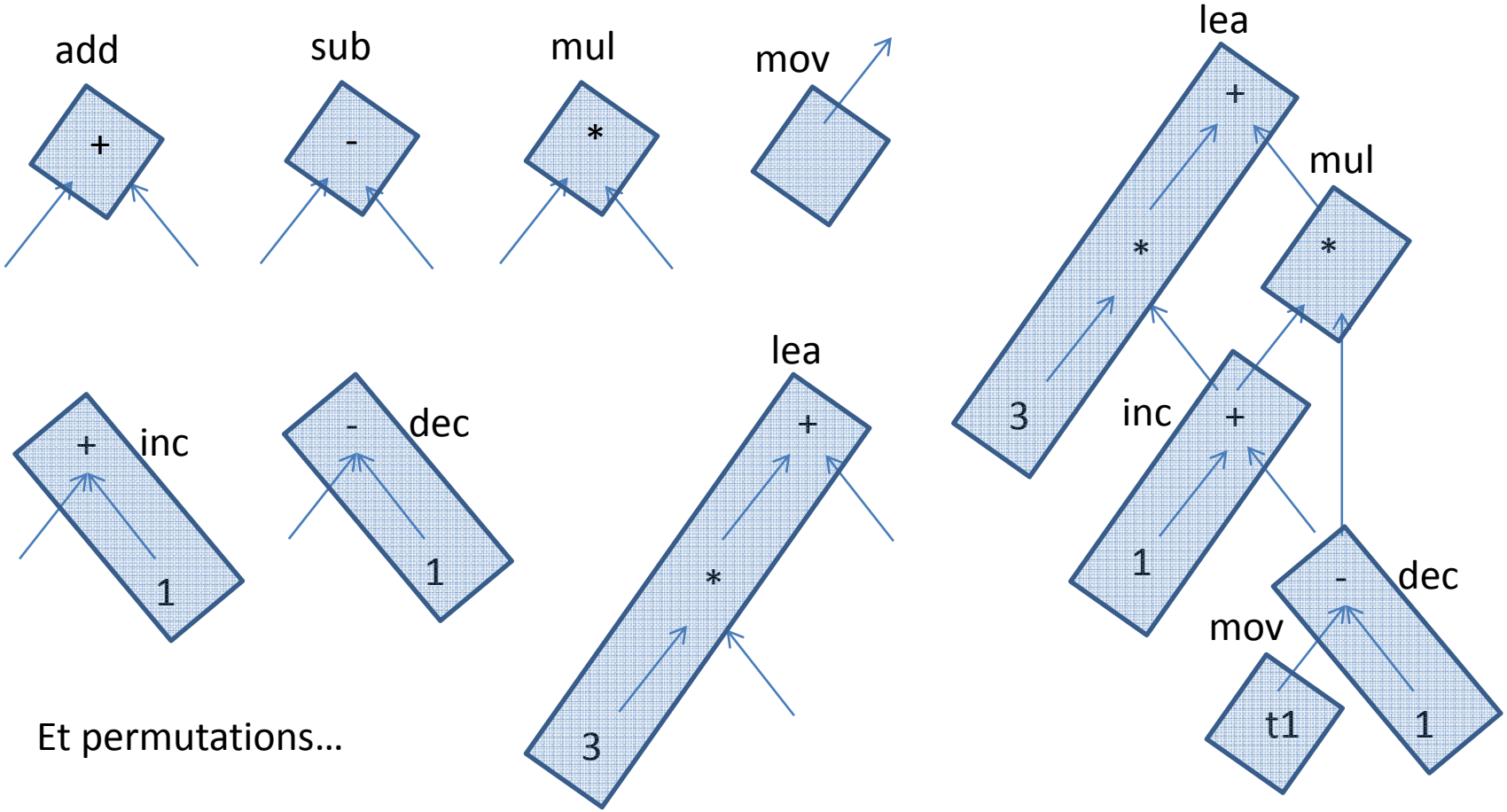


Et permutations…



Sélection des instructions

Trouver un pavage de coût minimum

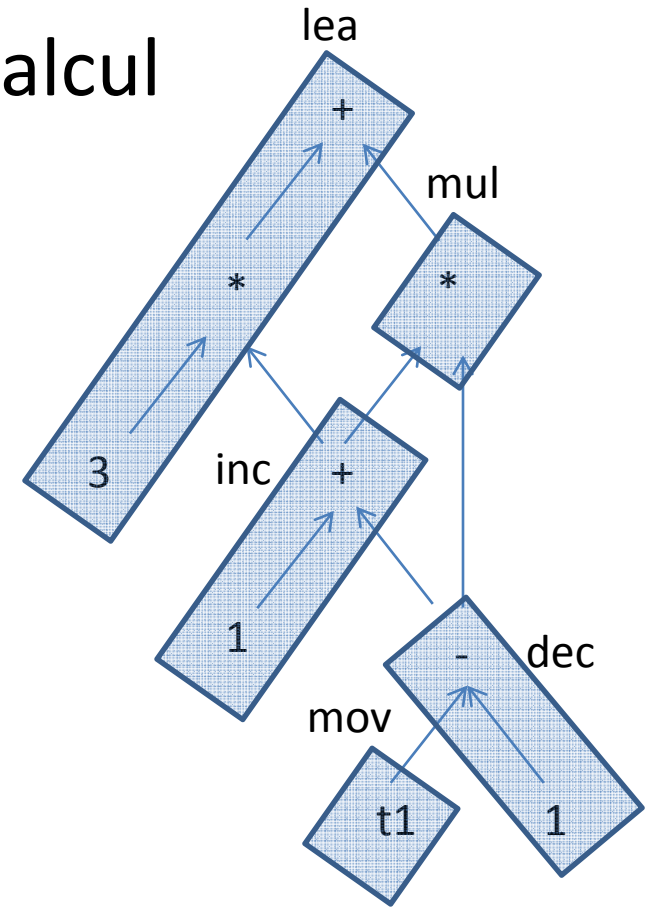


Modèle de coût

- Latence: #cycles total pour le calcul

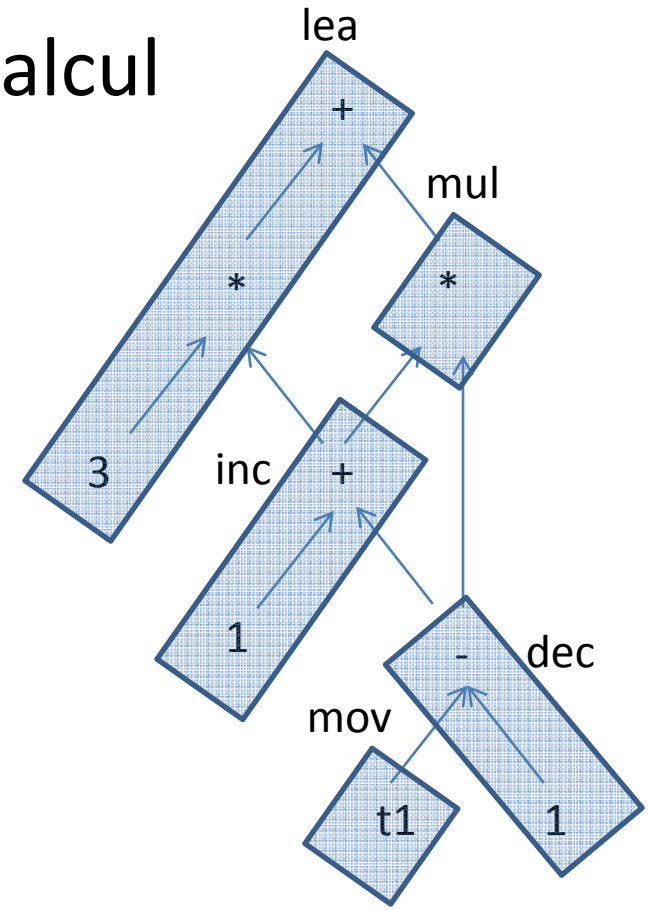
Instruction	Latence
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

X86-64 skylake µ



Modèle de coût

- Latence: #cycles total pour le calcul



Instruction	Latence
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

X86-64 skylake µ

dec	\	/	inc	\	/	mul				\	lea		
1	2	3	4	5	6	7	8	9	10	11	12	13	14

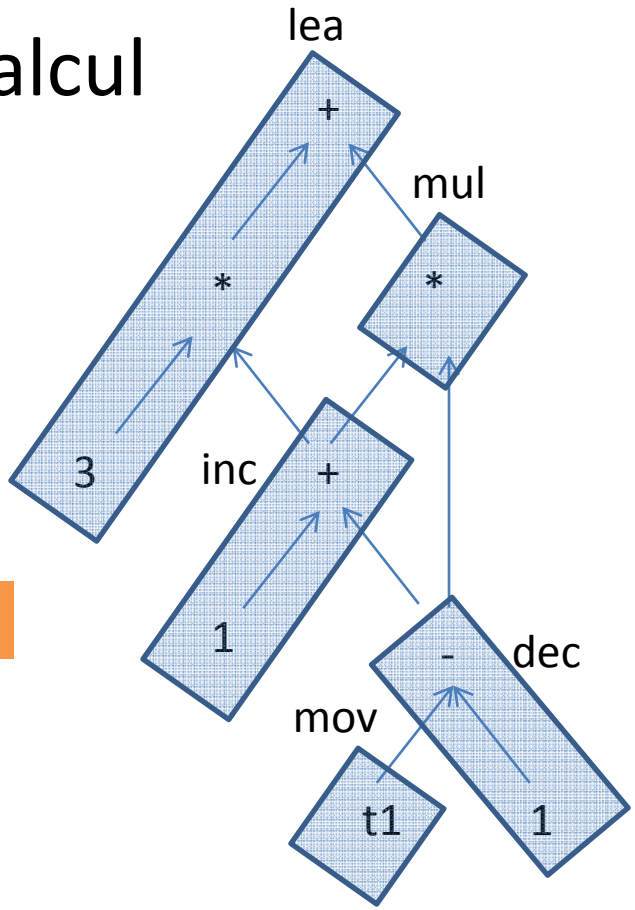
Modèle de coût

- Latence: #cycles total pour le calcul

Instruction	Latence
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

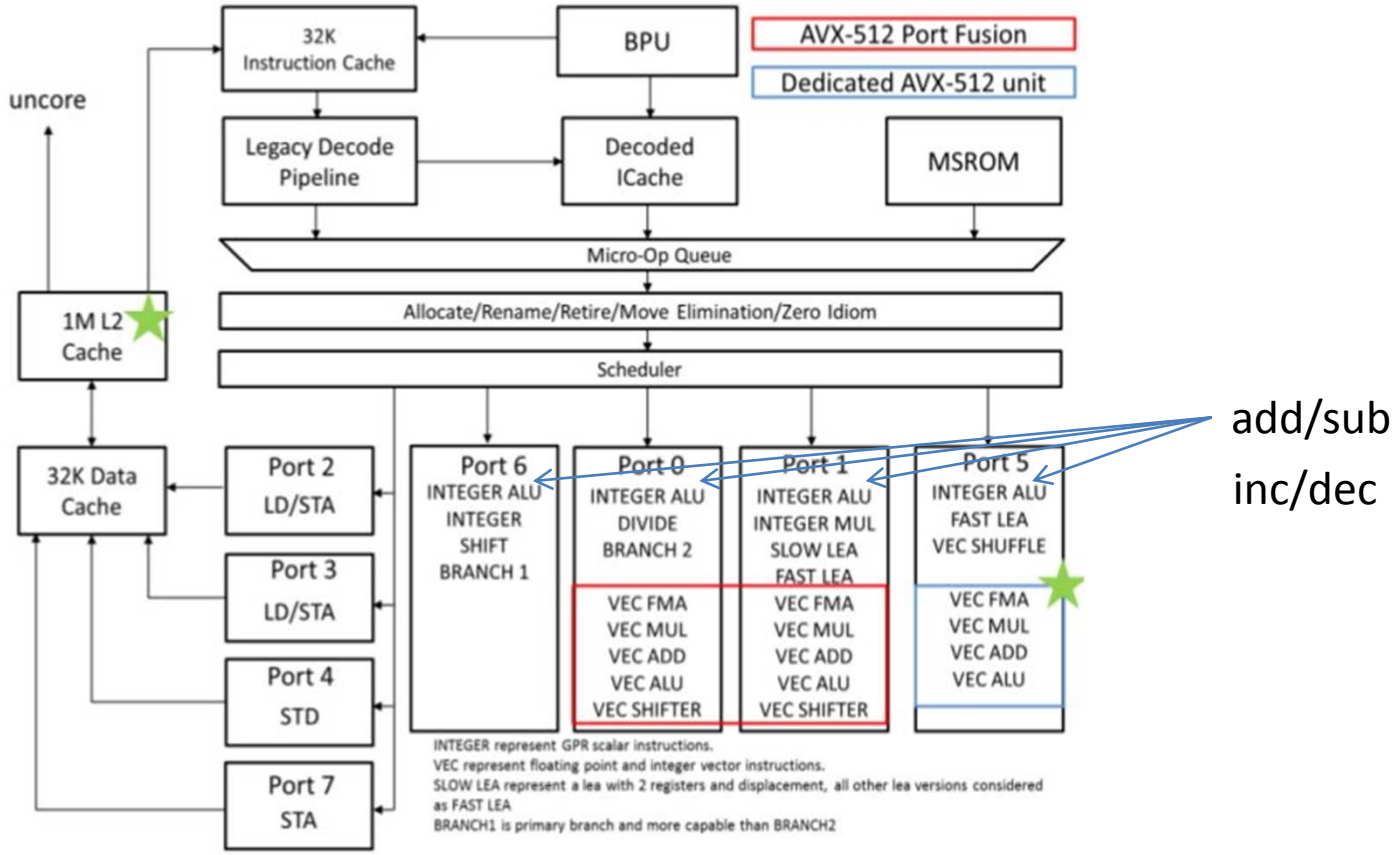
X86-64 skylake µ

Coût = somme des latences?



dec	\	/	inc	\	/	mul				\	lea		
1	2	3	4	5	6	7	8	9	10	11	12	13	14

Parallélisme d'instruction

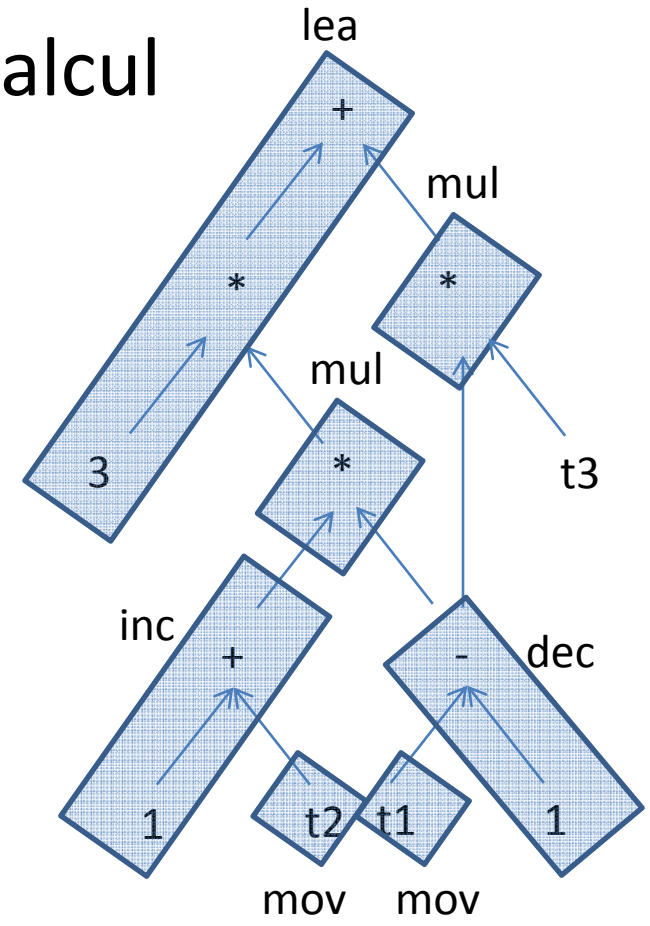


Skylake server microarchitecture, source: Intel

Modèle de coût

- Latence: #cycles total pour le calcul
- Débit: #cycles entre 2 inputs

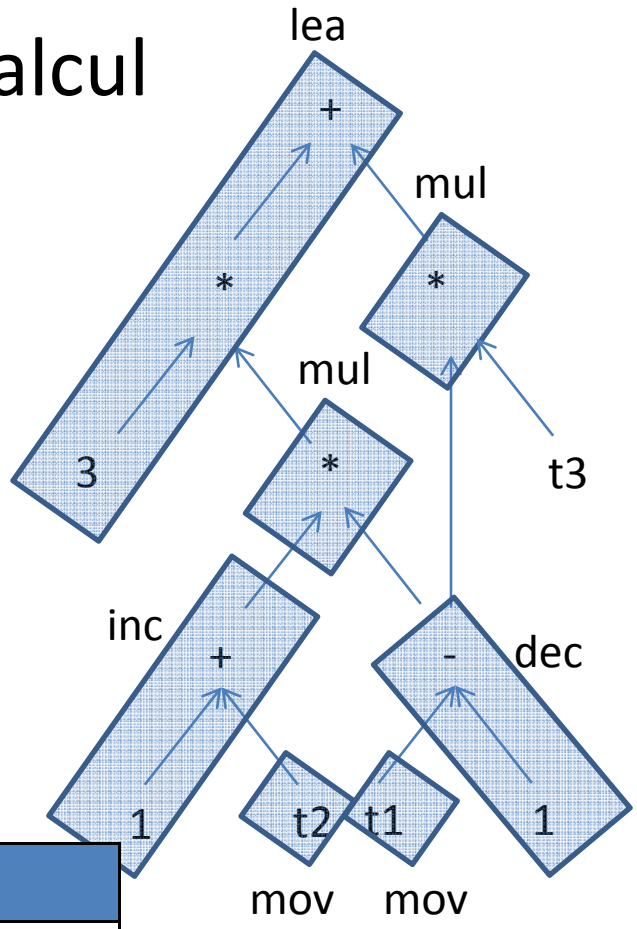
Instruction	Latence	Débit
mov	1	1
add/sub	1	0.25
dec/inc	1	0.25
mul	4	1
lea	3	1



Modèle de coût

- Latence: #cycles total pour le calcul
- Débit: #cycles entre 2 inputs

Instruction	Latence	Débit
mov	1	1
add/sub	1	0.25
dec/inc	1	0.25
mul	4	1
lea	3	1



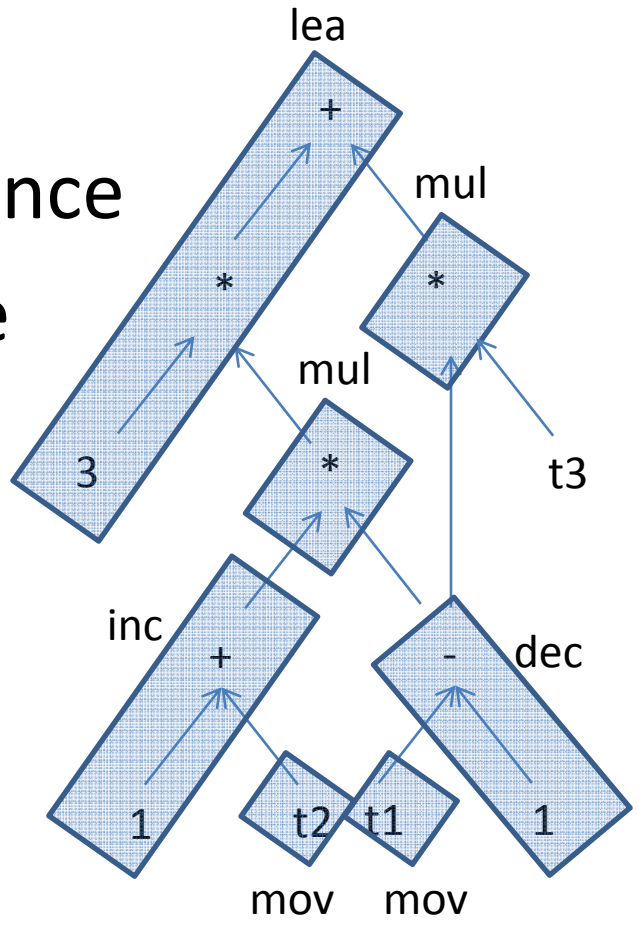
dec		\	/	mul				\	lea		
inc	/		mul				\				
1	2	3	4	5	6	7	8	9	10	11	12

Leçons

Le temps de calcul dépend de

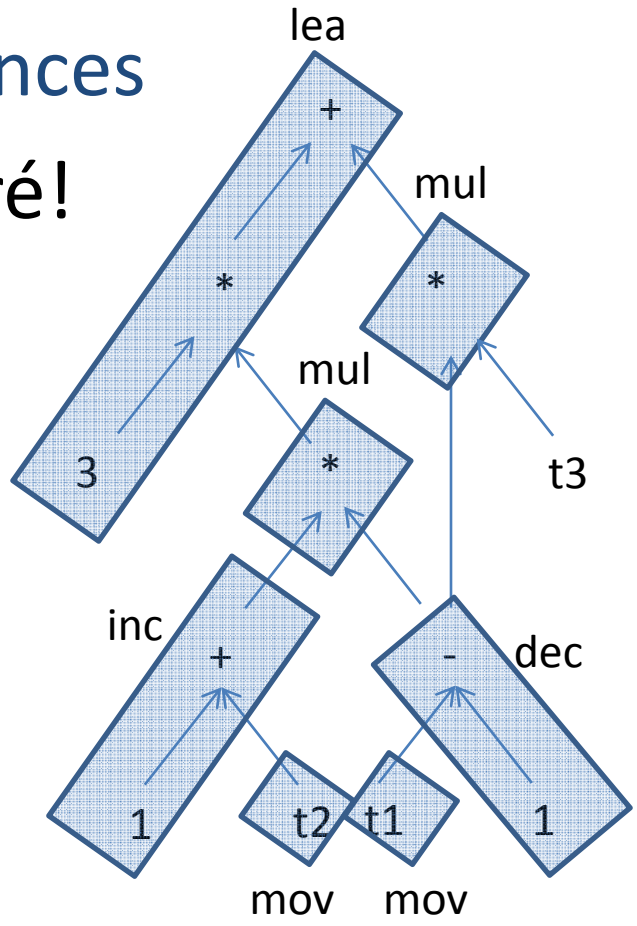
- Latence : instructions en séquence
- Débit : instructions en paralléle

Instruction	Latence	Débit
mov	1	1
add/sub	1	0.25
dec/inc	1	0.25
mul	4	1
lea	3	1



Modèle de coût

- Modèle usuel: somme des latences
- Parallélisme d'instruction ignoré!



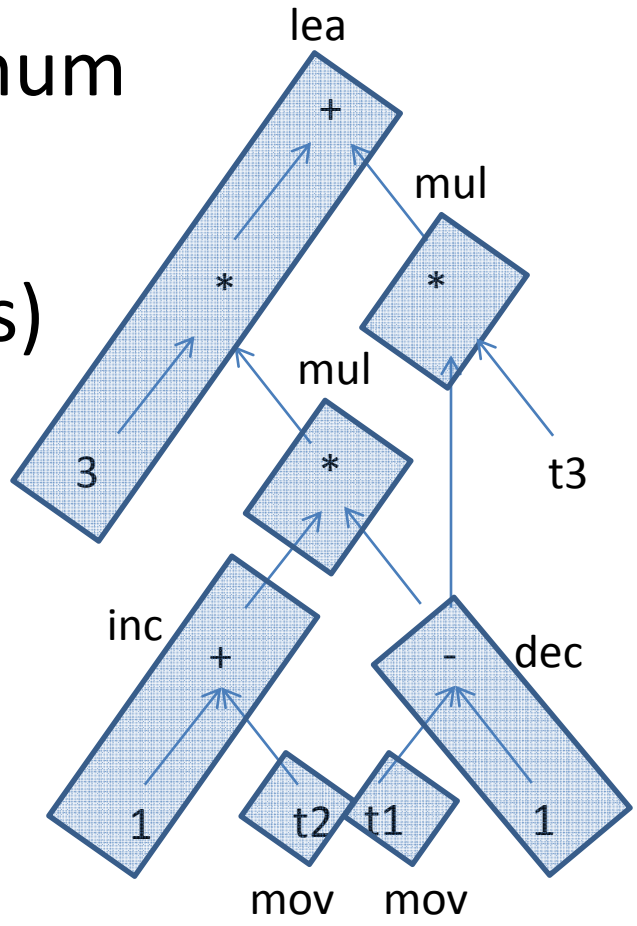
Instruction	Latence	Débit	Coût
mov	1	1	1
add/sub	1	0.25	1
dec/inc	1	0.25	1
mul	4	1	4
lea	3	1	3

Complexité

Trouver un pavage de coût minimum

- DAG: NP-complet
- Arbre: algorithme en O(Noeuds)

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3



Programmation dynamique

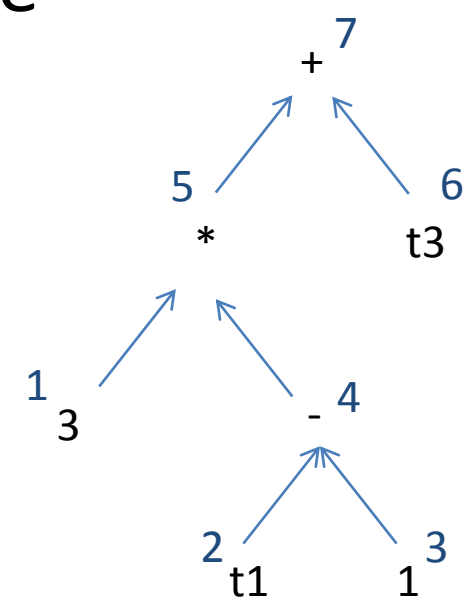
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1		
2		
3		
4		
5		
6		
7		



Programmation dynamique

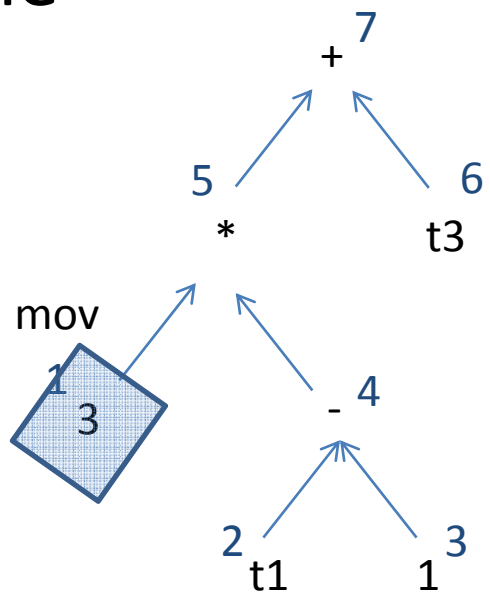
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2		
3		
4		
5		
6		
7		



Programmation dynamique

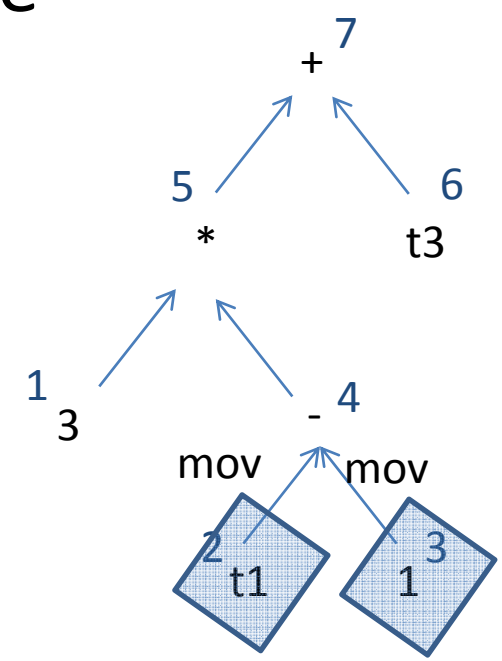
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4		
5		
6		
7		



Programmation dynamique

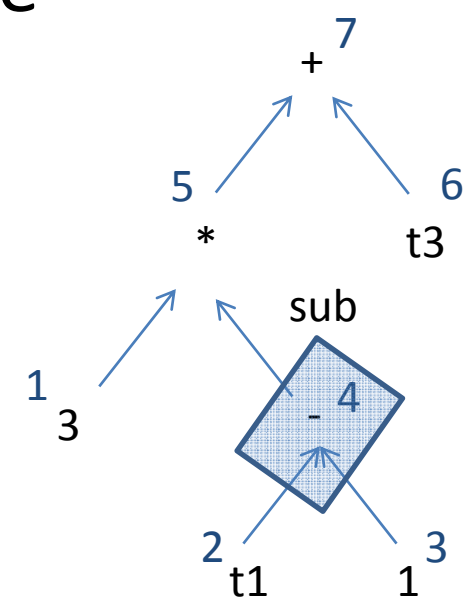
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	sub?	
5		
6		
7		



Programmation dynamique

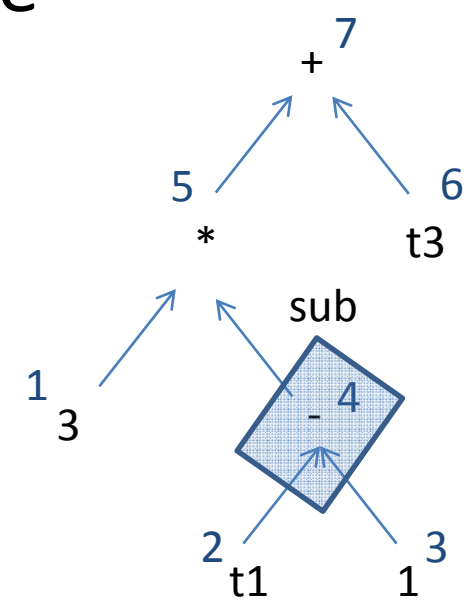
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	sub?	1+1+1 = 3
5		
6		
7		



Programmation dynamique

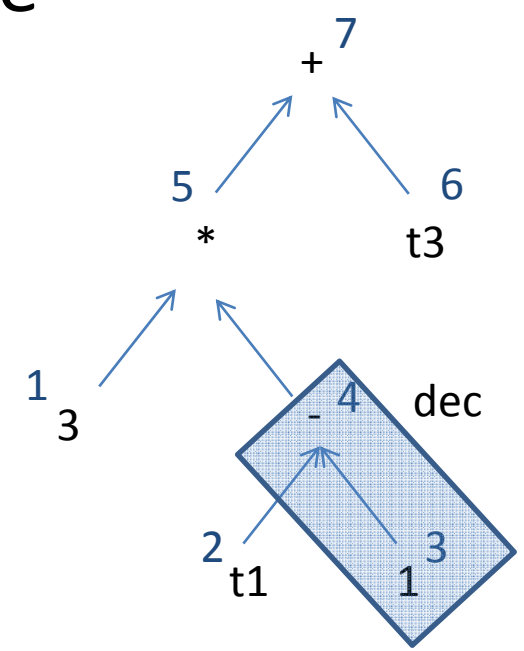
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec?	
5		
6		
7		



Programmation dynamique

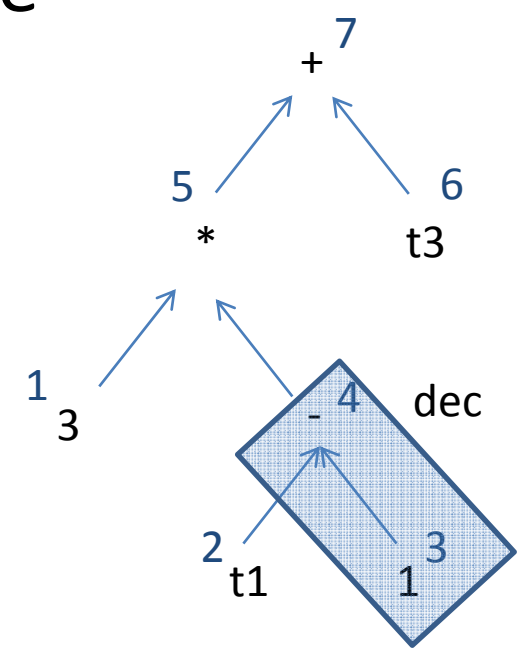
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec?	1+1 = 2
5		
6		
7		



Programmation dynamique

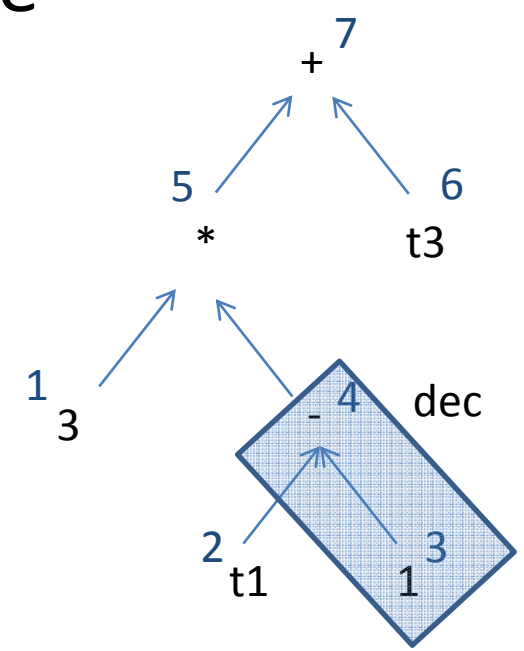
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5		
6		
7		



Programmation dynamique

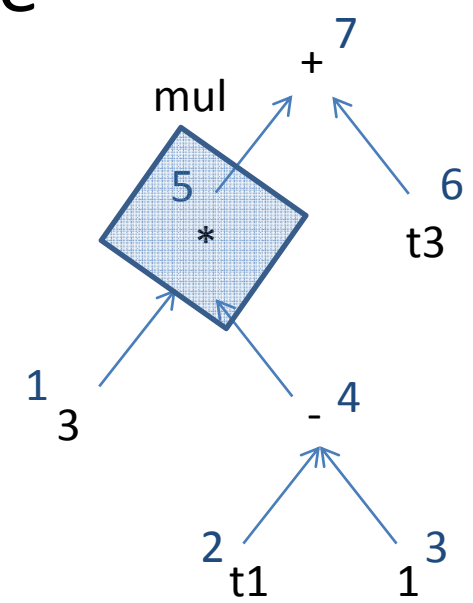
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	
6		
7		



Programmation dynamique

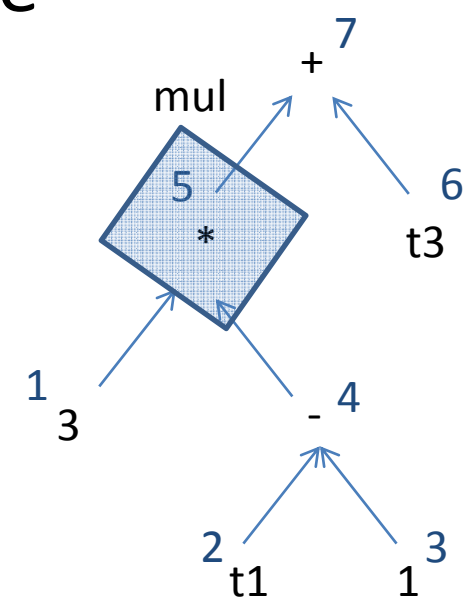
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6		
7		



Programmation dynamique

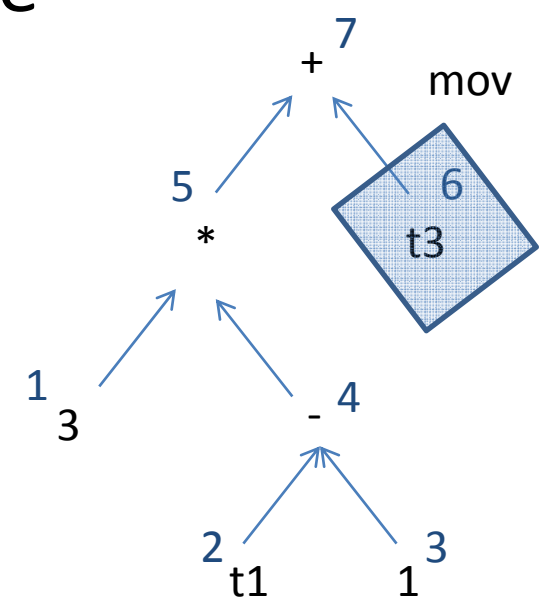
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	mov	1
7		



Programmation dynamique

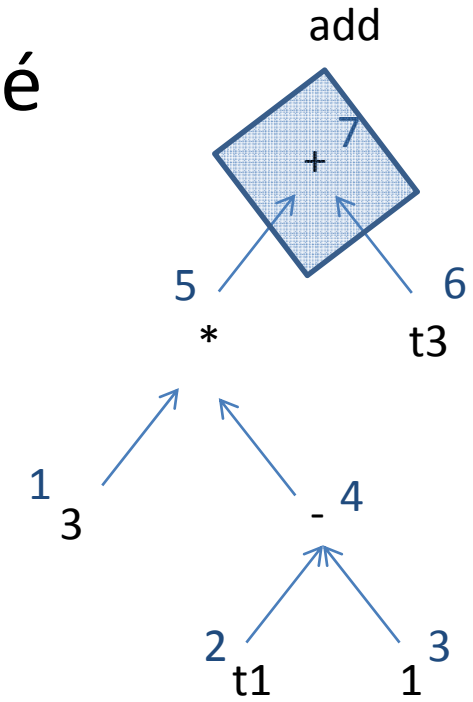
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	mov	1
7	add?	1+7+1 = 9



Programmation dynamique

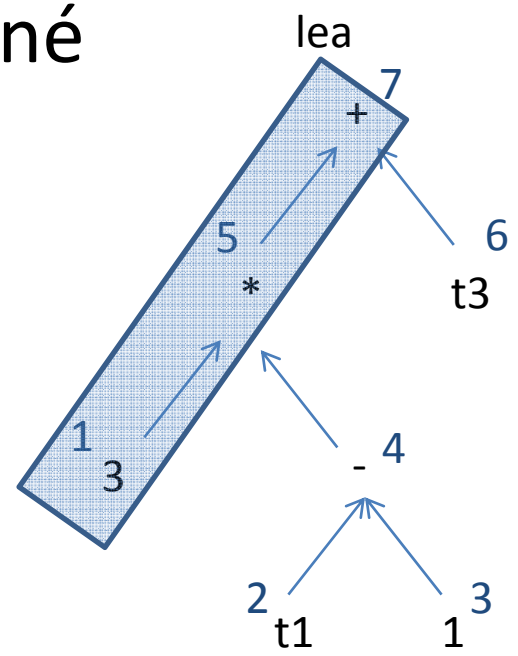
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	mov	1
7	lea?	3+2+1 = 6



Programmation dynamique

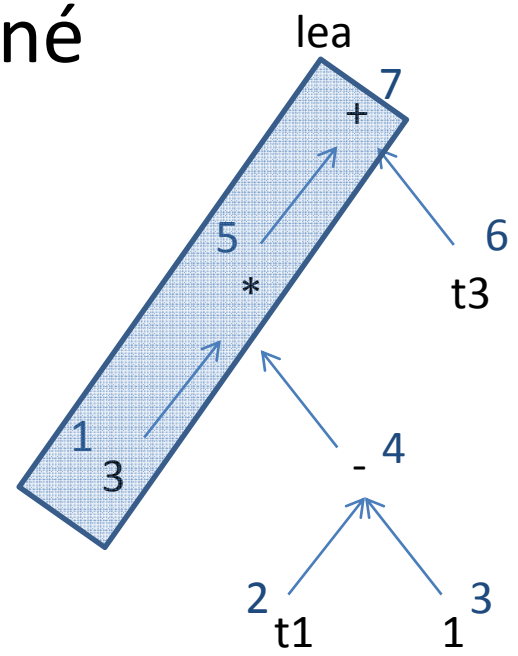
Dans l'ordre post-fixe, on construit:

i_opt: noeud --> instruction

c_opt: noeud --> coût total enraciné

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	mov	1
7	lea	3+2+1 = 6

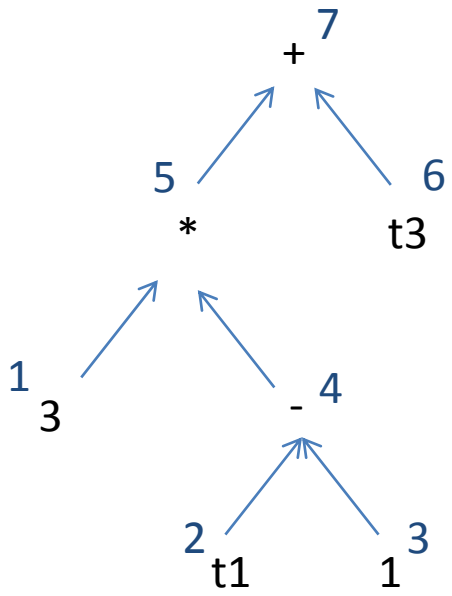


Programmation dynamique

On garde la sélection optimale
en partant de la racine

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	mov	1
7	lea	3+2+1 = 6

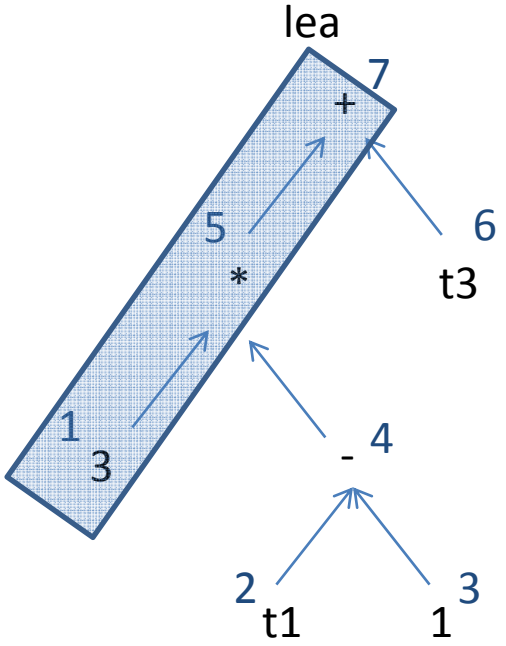


Programmation dynamique

On garde la sélection optimale
en partant de la racine

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	mov	1
7	**lea**	**3+2+1 = 6**

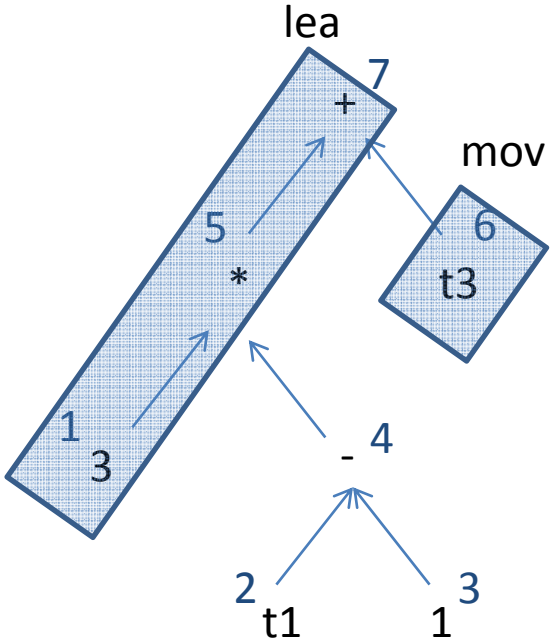


Programmation dynamique

On garde la sélection optimale
en partant de la racine

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	dec	1+1 = 2
5	mul	4+1+2 = 7
6	**mov**	**1**
7	**lea**	**3+2+1 = 6**

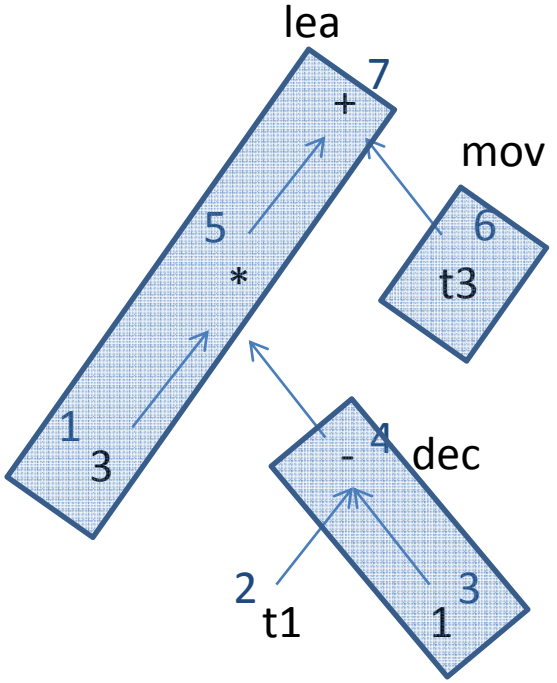


Programmation dynamique

On garde la sélection optimale
en partant de la racine

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	mov	1
3	mov	1
4	**dec**	**1+1 = 2**
5	mul	4+1+2 = 7
6	**mov**	**1**
7	**lea**	**3+2+1 = 6**

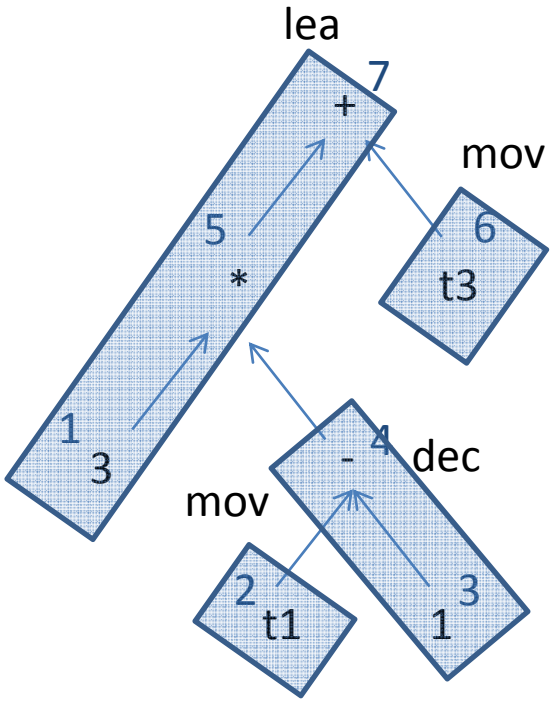


Programmation dynamique

On garde la sélection optimale
en partant de la racine

Pavé	Coût
mov	1
add/sub	1
dec/inc	1
mul	4
lea	3

n	i_opt(n)	c_opt(n)
1	mov	1
2	**mov**	**1**
3	mov	1
4	**dec**	**1+1 = 2**
5	mul	4+1+2 = 7
6	**mov**	**1**
7	**lea**	**3+2+1 = 6**



Algorithme

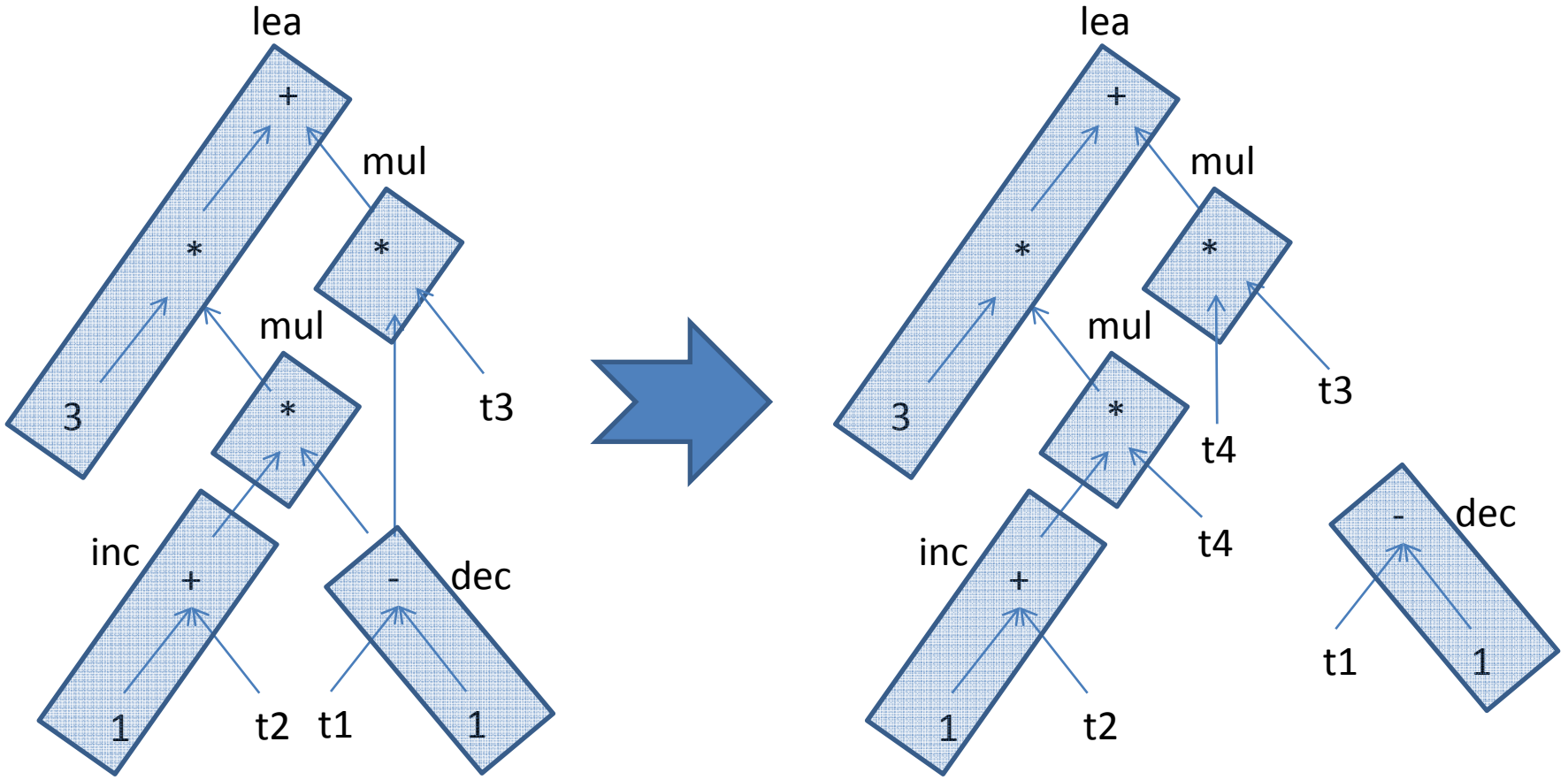
1. Pour chaque noeud dans l'ordre postfixe:
   Trouver l'instruction qui minimise
   coût instruction + coût des sous-arbres

2. Sélectionner l'instruction optimale à la racine et récursivement sur les sous-arbres.

Complexité: O(Noeuds)

Traitement des DAGs “purs”

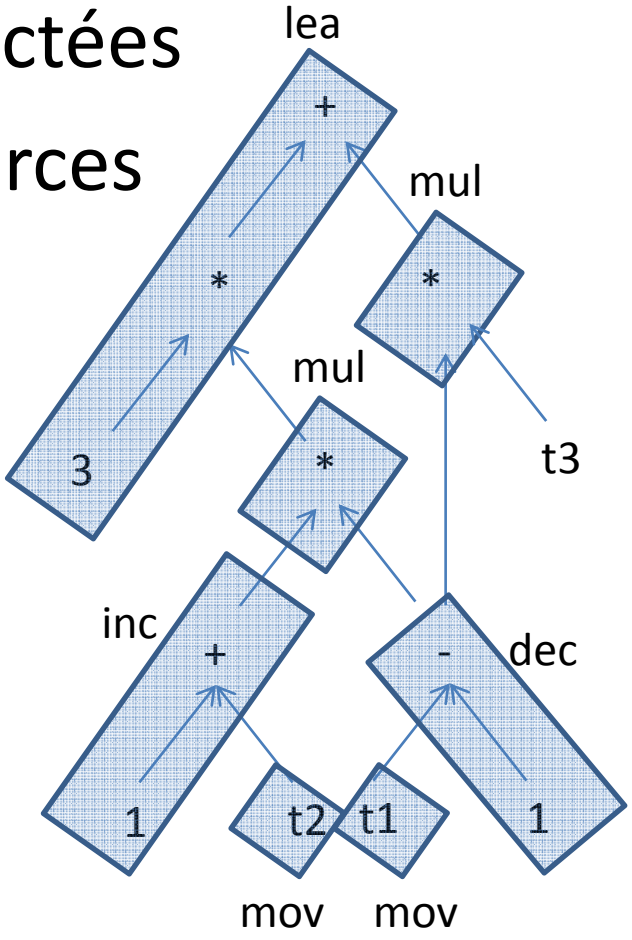
Décrocher les noeuds partagés



Ordonnancement

Dans quel ordre exécuter les instructions?

- Correction: dépendences respectées
- Efficacité: utilisation des ressources
  - Parallélisme d'instruction
  - Pipeline
  - Registres



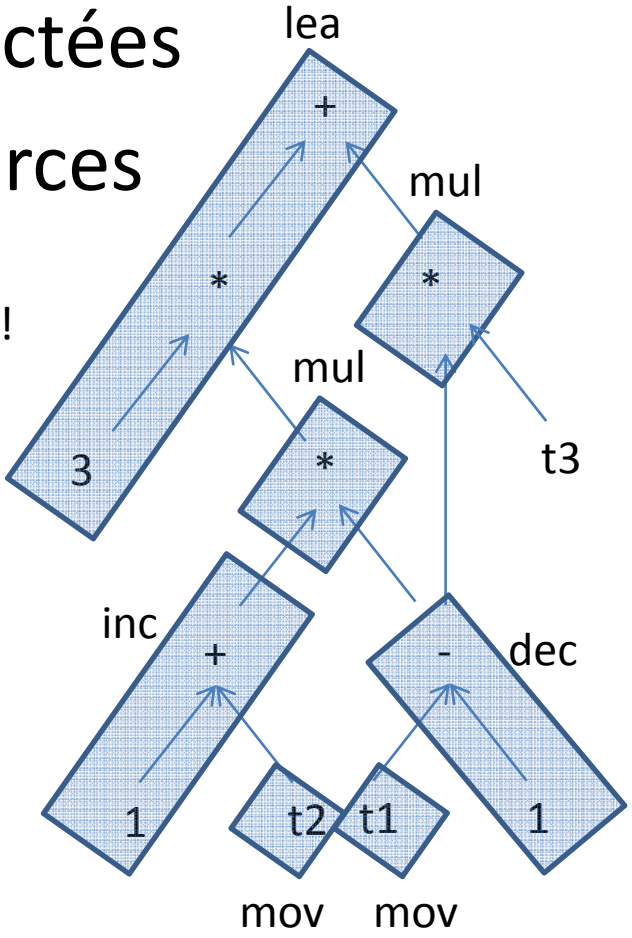
Ordonnancement

Dans quel ordre exécuter les instructions?

- Correction: dépendences respectées
- Efficacité: utilisation des ressources
  - Parallélisme d'instruction
  - Pipeline

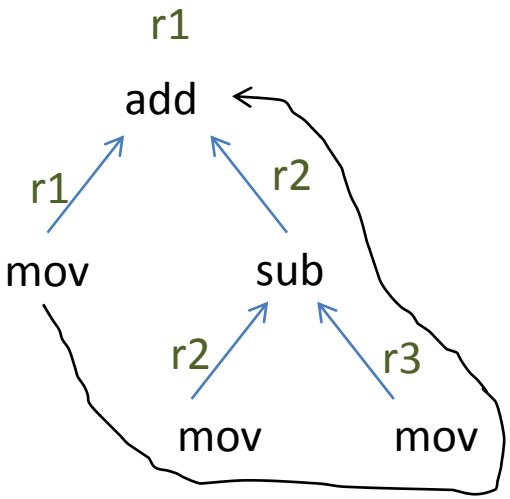
Par le CPU!

  - Registres

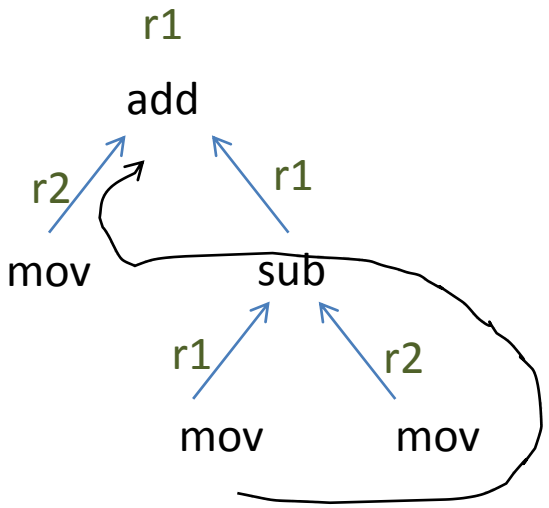


Ordonnancement

On cherche un ordre d'exécution qui minimise le besoin en registres



3 registres



2 registres

Complexité

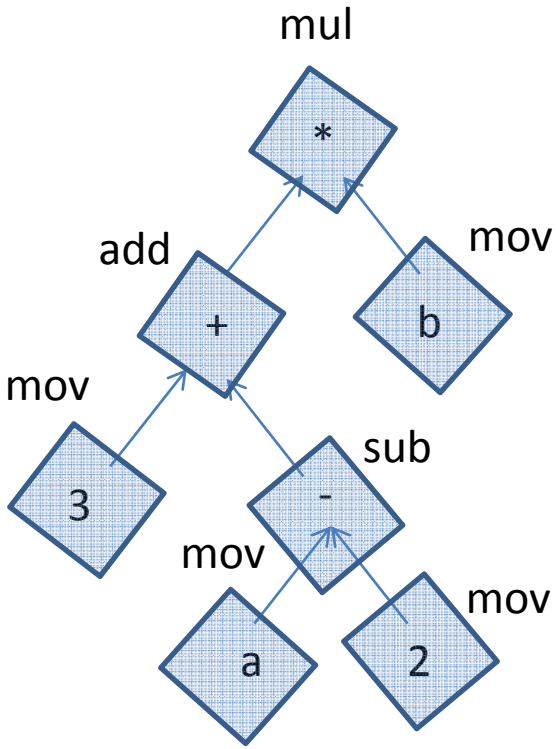
Trouver un ordre qui minimise le besoin en registre est:

- DAG: NP-complet
- Arbres: algorithme en O(Noeuds)

Stratégie: décomposition du DAG en sous-arbres puis ordonnancement des arbres

Algorithme de Sethi-Ullman

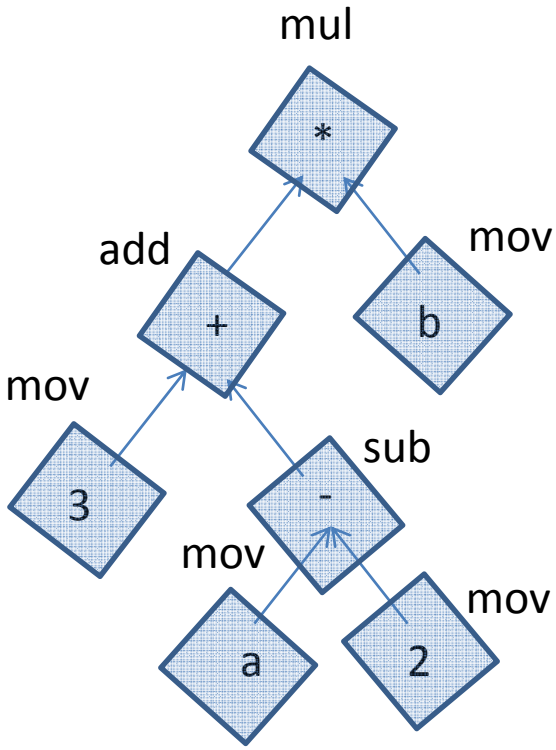
rg: noeud --> #registres requis



Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

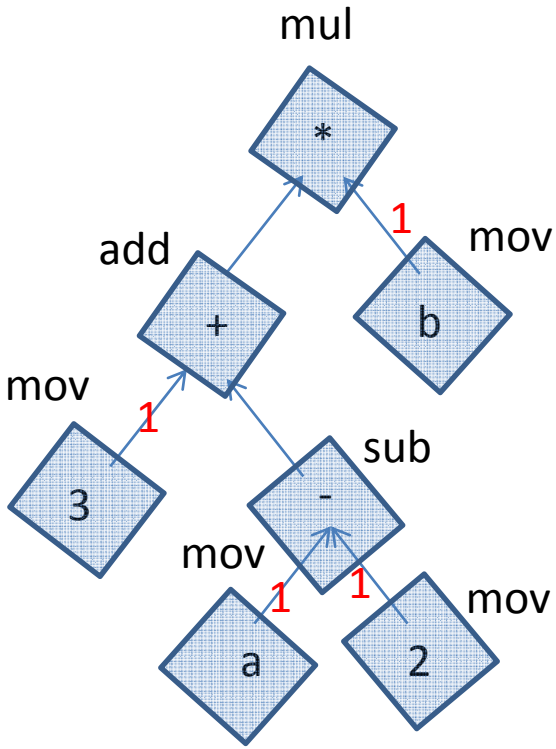
- rg(feuille) = ?



Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

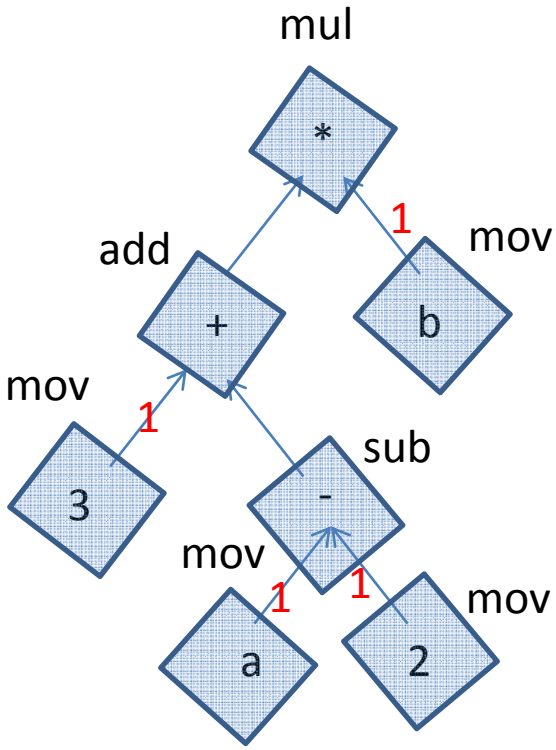
- rg(feuille) = 1



Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

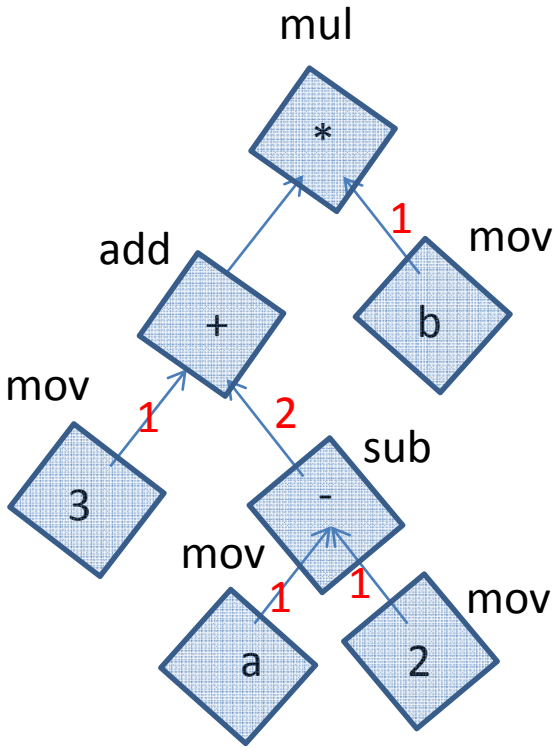
- rg(feuille) = 1
- rg(op(n1,n2)) =
  - rg(n1) = rg(n2): ?



Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

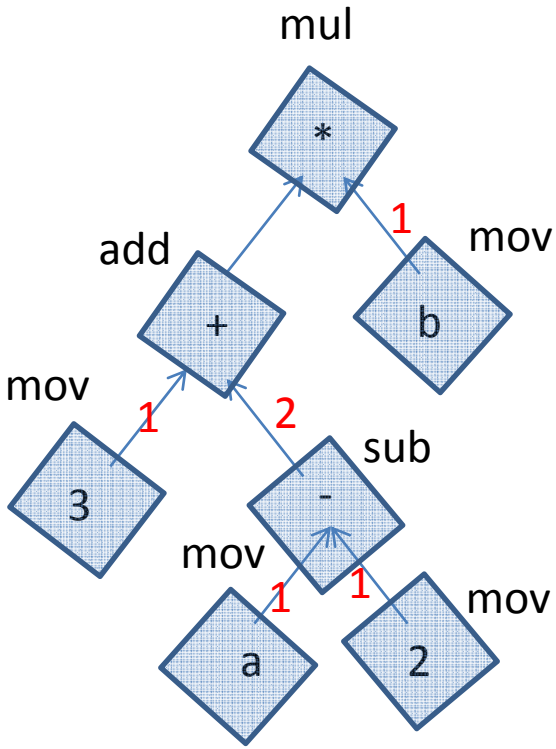
- rg(feuille) = 1
- rg(op(n1,n2)) =
  - rg(n1) = rg(n2): 1+rg(n1)



Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

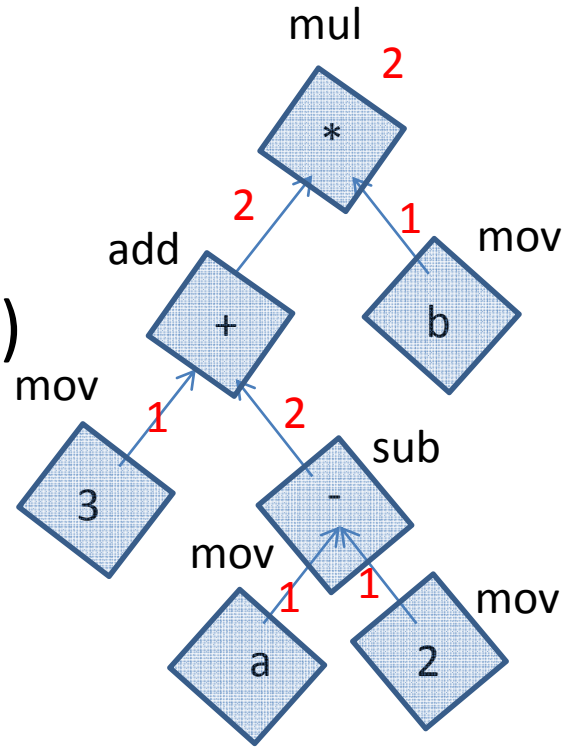
- rg(feuille) = 1
- rg(op(n1,n2)) =
  - rg(n1) = rg(n2): 1+rg(n1)
  - rg(n1) != rg(n2): ?



Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

- rg(feuille) = 1
- rg(op(n1,n2)) =
  - rg(n1) = rg(n2): 1+rg(n1)
  - rg(n1) != rg(n2): max(rg(n1),rg(n2))



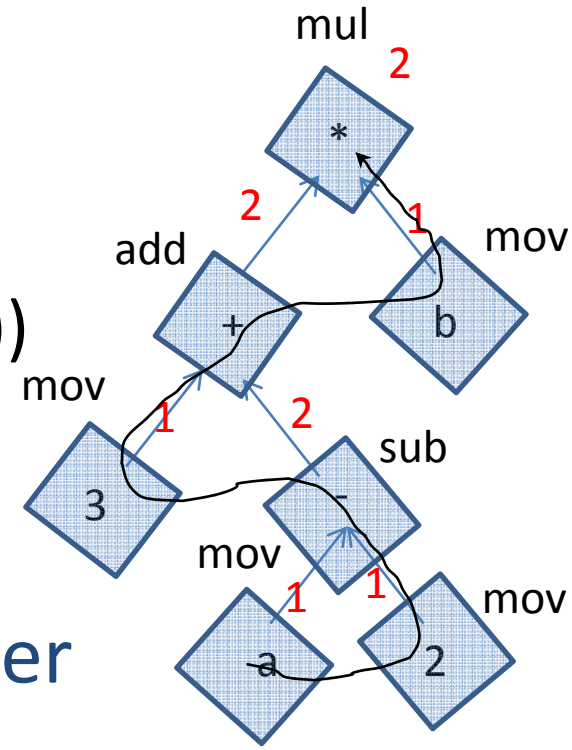
Algorithme de Sethi-Ullman

rg: noeud --> #registres requis

- rg(feuille) = 1
- rg(op(n1,n2)) =
  - rg(n1) = rg(n2): 1+rg(n1)
  - rg(n1) != rg(n2): max(rg(n1),rg(n2))

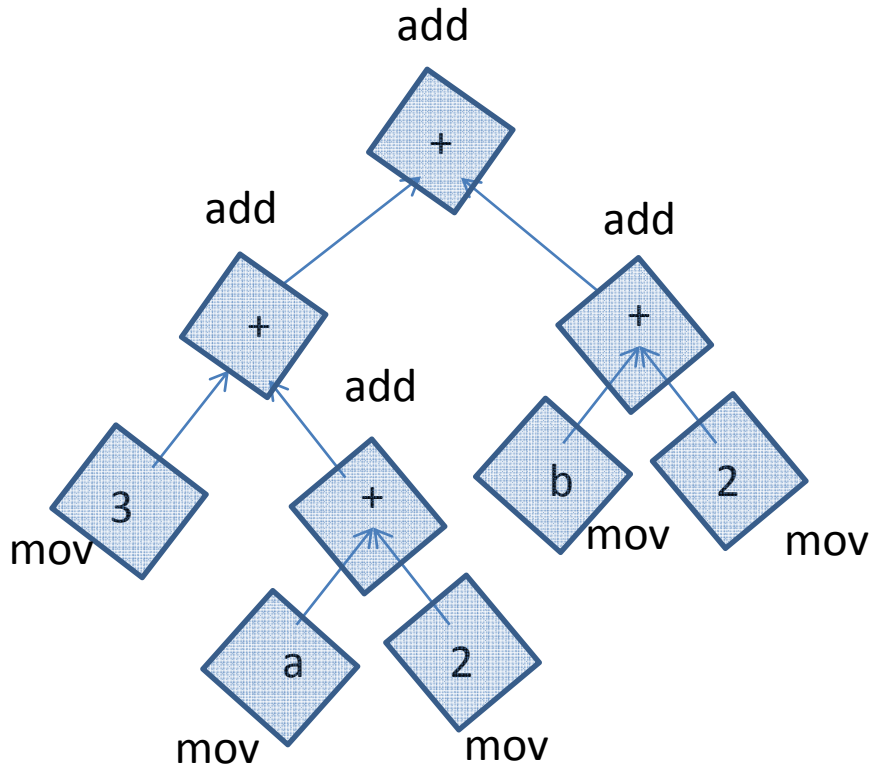
Ordre postfixe

+ rg le plus grand évalué en premier



Quizz

Trouver un ordonnancement optimal pour:



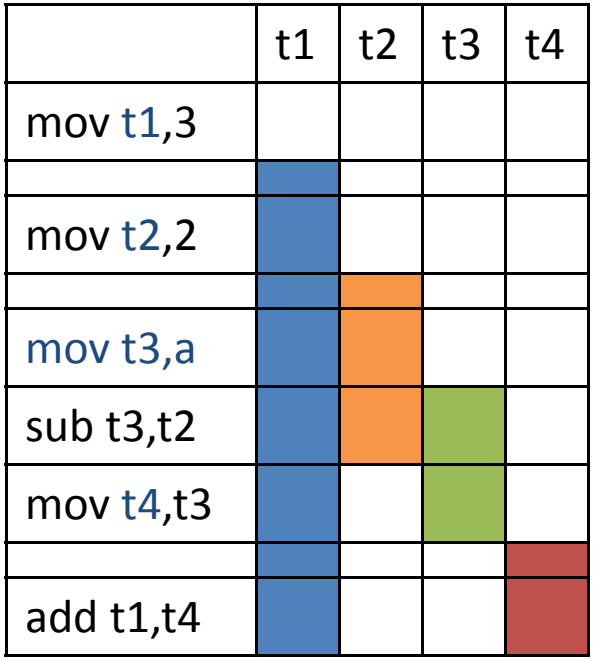
Allocation des registres

- Allocation des ressources de stockage (registres, pile) aux temporaires.
  - alloc: t --> { r1, …, rK, slot1, … }
- Priorité aux registres
- On fixe le alloc ? avec le nombre de slots de pile alloués

```
mov t1,3

mov t2,2

mov t3,a
sub t3,t2
mov t4,t3

add t1,t4
```

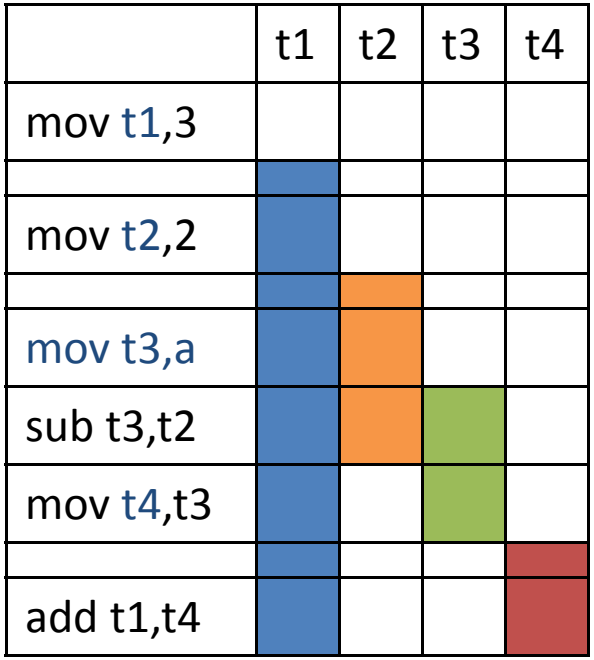
Interférences

- t1 et t2 interfèrent (t1 |\| t2) s'ils vivent en même temps en un point d'exécution.



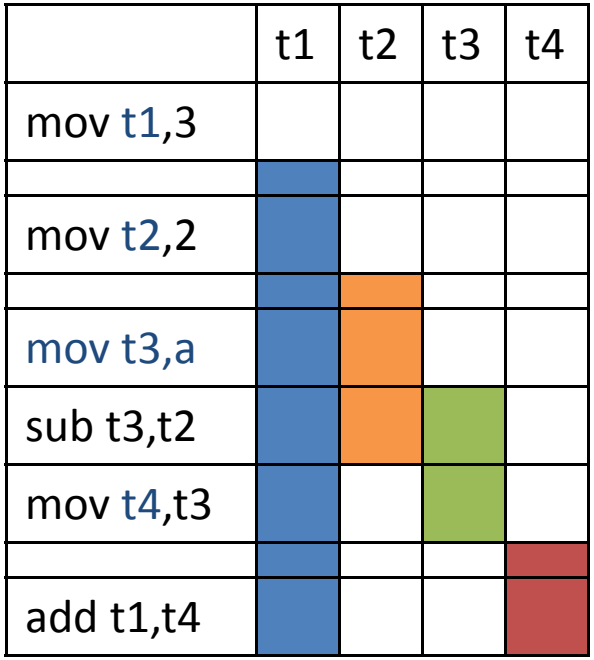
Interférences

- t1 et t2 interfèrent (t1 |\| t2) s'ils vivent en même temps en un point d'exécution.
- Correction:
  t1 |\| t2 => alloc(t1) != alloc(t2)



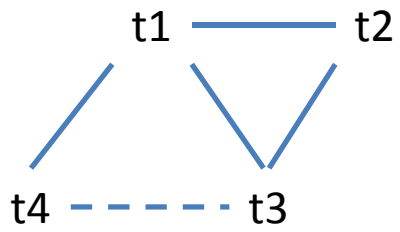
Interférences

- t1 et t2 interfèrent (t1 |\| t2) s'ils vivent en même temps en un point d'exécution.
- Correction:
  t1 |\| t2 => alloc(t1) != alloc(t2)
- Efficacité: autant que possible,
  - Allouer un registre
  - Pour une copie mov t1,t2:
    alloc(t1) = alloc(t2)



Graphe d'interférence

- Noeuds: temporaires
- Arc d'interférence t1 – t2 ssi t1 |\| t2
- Arc d'affinité t1 - - - t2 ssi il existe mov t1,t2

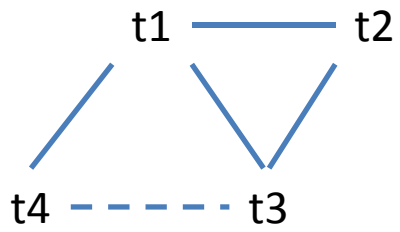


	t1	t2	t3	t4
mov t1,3				
mov t2,2				
mov t3,a				
sub t3,t2				
mov t4,t3				
add t1,t4				

Graphe d'interférence

- Noeuds: temporaires
- Arc d'interférence t1 – t2 ssi t1 |\| t2
- Arc d'affinité t1 - - - t2 ssi il existe mov t1,t2

Allocation = K-coloriage!

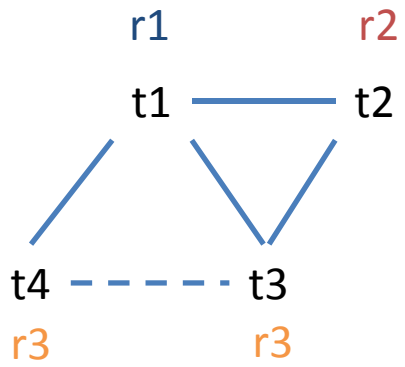


	t1	t2	t3	t4
mov t1,3				
mov t2,2				
mov t3,a				
sub t3,t2				
mov t4,t3				
add t1,t4				

Graphe d'interférence

- Noeuds: temporaires
- Arc d'interférence t1 – t2 ssi t1 |\| t2
- Arc d'affinité t1 - - - t2 ssi il existe mov t1,t2

Allocation = K-coloriage!



	t1	t2	t3	t4
mov t1,3				
mov t2,2				
mov t3,a				
sub t3,t2				
mov t4,t3				
add t1,t4				

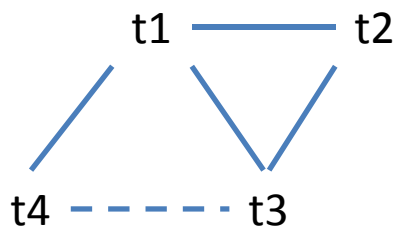
Complexité

- Le problème du K-coloriage est NP-complet pour K >= 3 couleurs
- Tout graphe non-orienté peut être interprété comme un graphe d'interférence
- Le problème de l'allocation de registres est NP-complet pour K >= 3 registres

Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

- Colorier(G – {*n*})
- Allouer une couleur restante à *n*

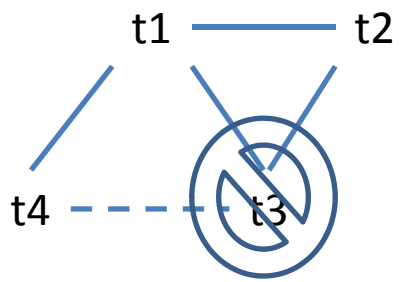


K = 3 registres

Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



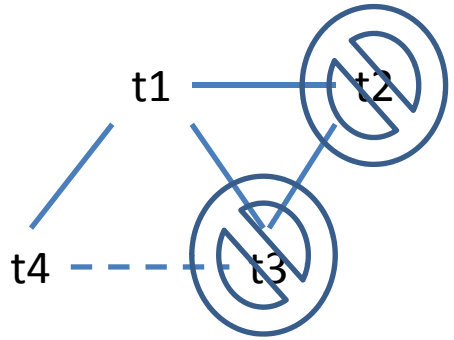
K = 3 registres

n : t3

Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



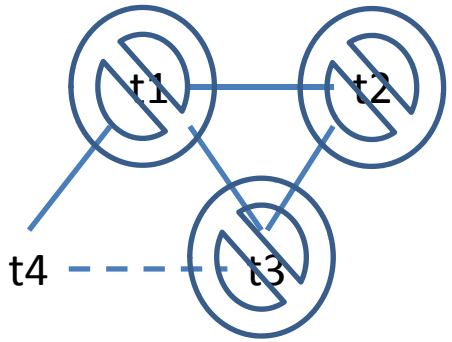
K = 3 registres

n : t3, t2

Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



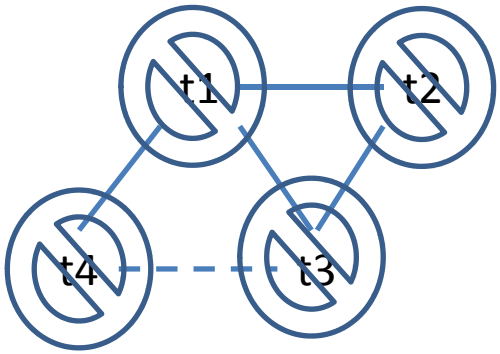
K = 3 registres

n : t3, t2, t1

Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



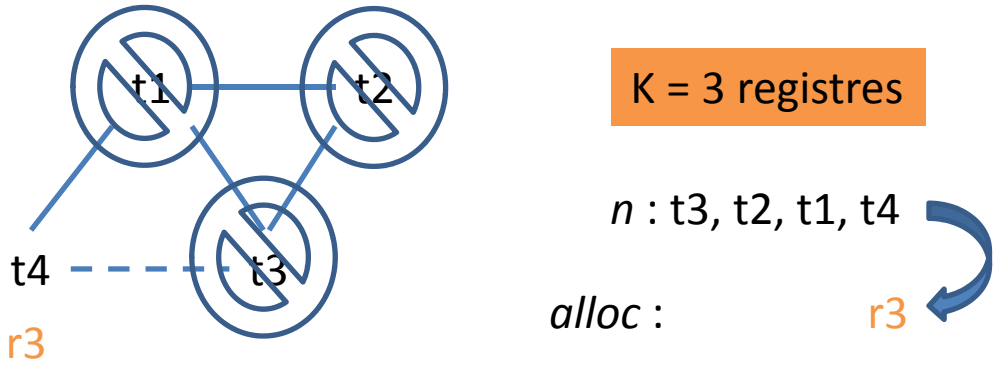
K = 3 registres

n : t3, t2, t1, t4

Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

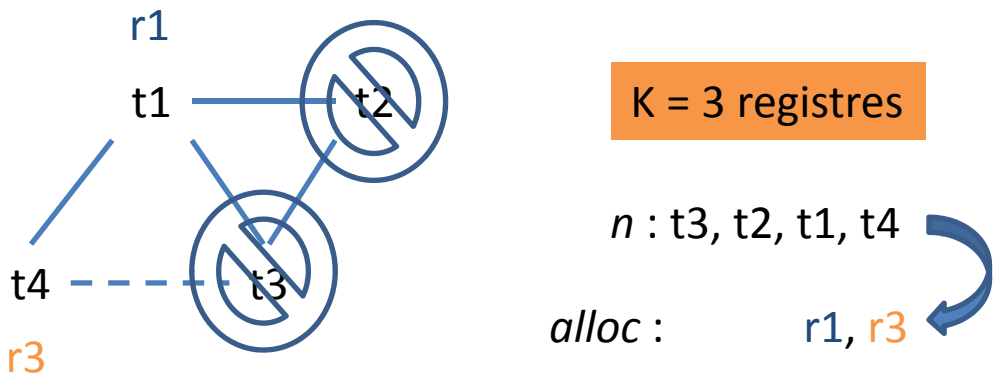
- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

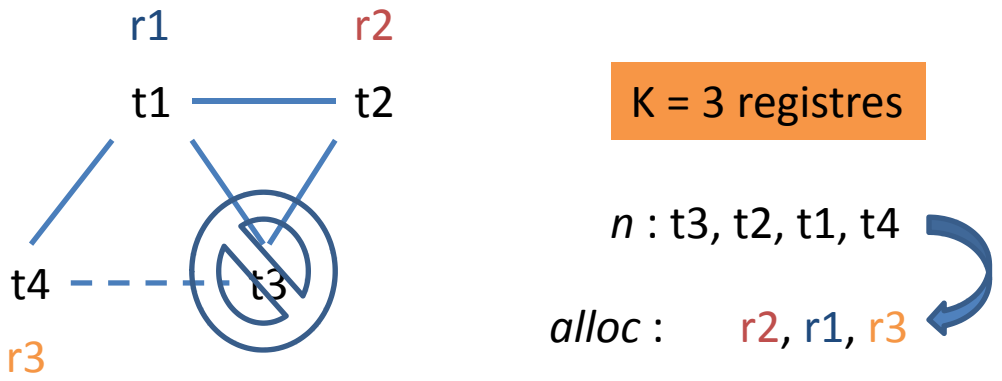
- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

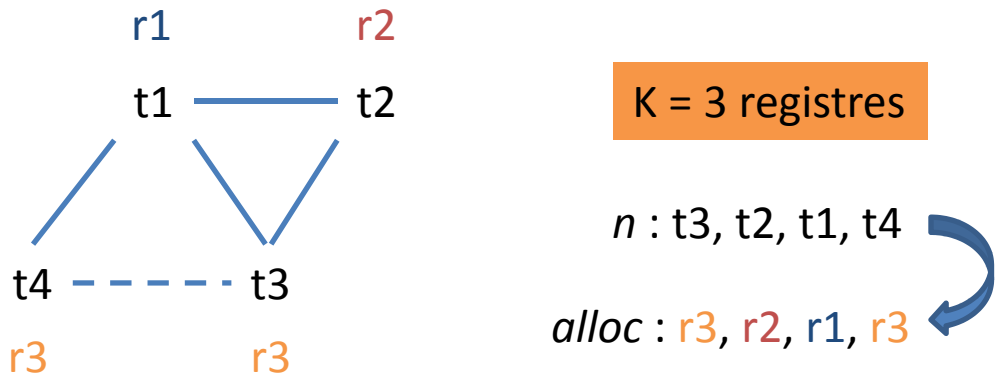
- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



Coloriage par simplification

Un noeud *n* est trivialement K-colorable s'il a moins de K voisins

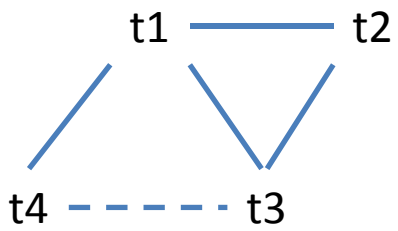
- Colorier(G – {*n*})
- Allouer une couleur restante à *n*



Coloriage par simplification

Si aucun noeud est trivialement K-colorable:

- Choisir un noeud *n* de plus fort degré
- Colorier(G – {*n*})
- S'il reste une couleur, alors l'allouer
  Sinon: **marquer n**; retirer n



K = 2 registres

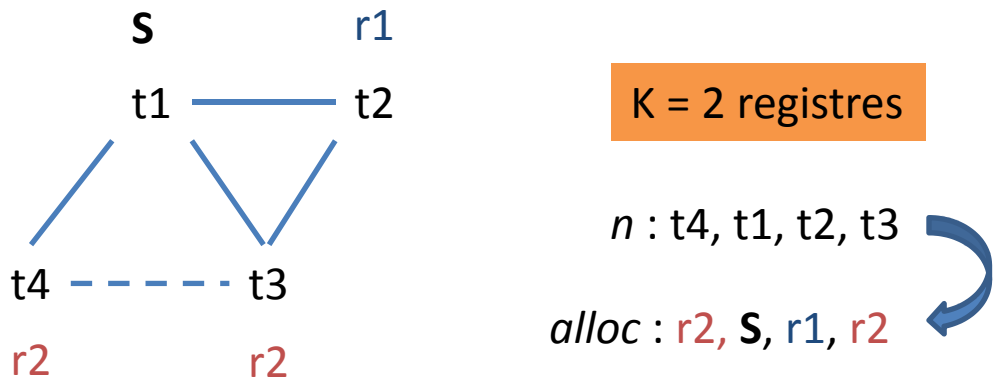
n : t4, t1, t2, t3

alloc :

Coloriage par simplification

Si aucun noeud est trivialement K-colorable:

- Choisir un noeud *n* de plus fort degré
- Colorier(G – {*n*})
- S'il reste une couleur, alors l'allouer
  Sinon: **marquer n**; retirer n



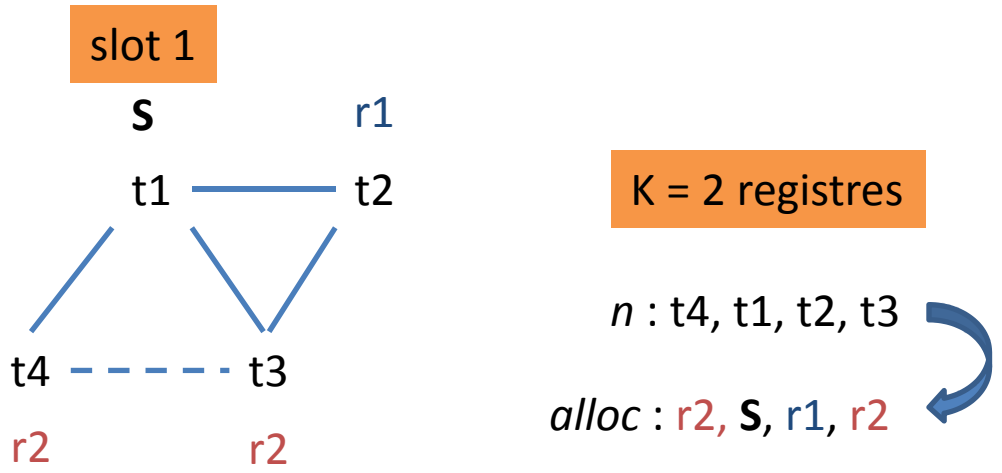
Algorithme de Chaitin

Colorier(G,K)

- Si il existe un noeud n avec moins de K voisins:
  - Colorier(G – {*n*})
  - Allouer une couleur restante à *n*
- Sinon si aucun noeud est trivialement K-colorable:
  - Choisir un noeud *n* de plus fort degré
  - Colorier(G – {*n*})
  - S'il reste une couleur, alors l'allouer
    Sinon: **marquer *n***; retirer *n*

Allocation des slots de pile

- Il suffit de colorier le graphe restreint aux noeuds marqués avec K = +oo slots
- La couleur donne le décalage dans la pile



Vidage

- Avant chaque lecture de t1, charger le slot 1 dans un registre de vidage
- Après chaque écriture de t1, sauvegarder le registre de vidage dans le slot 1

add t1,t4 --> mov r,[slot1]
add r,r2
mov [slot1],r

mov t1,3

mov t2,2

mov t3,a

sub t3,t2

mov t4,t3

add t1,t4

Vidage

- Opérations binaires
  => au plus 2 registres de vidage r et r':
  add t1,t4 --> mov r,[slot1]
  mov r',[slot4]
  add r,r'
  mov [slot1],r
- Comment allouer les registres de vidage?

mov t1,3

mov t2,2

mov t3,a

sub t3,t2

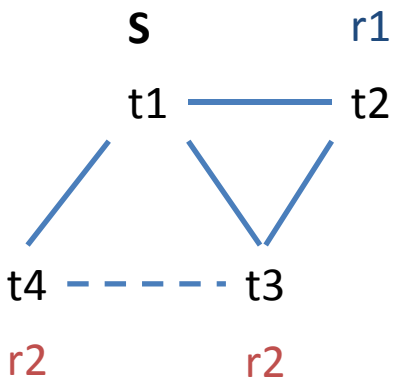
mov t4,t3

add t1,t4

Coloriage itératif

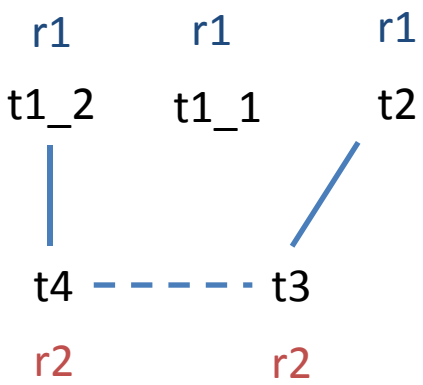
- Quand des noeuds sont marqués, on itére:
  - On produit le code avec le vidage (ici t1)
  - On recommence l'allocation

Avant itération:



	t1_1	t1_2	t2	t3	t4
mov t1_1,3					
mov [slot],t1_1					
mov t2,2					
mov t3,a					
sub t3,t2					
mov t4,t3					
mov t1_2,[slot]					
add t1_2,t4					
mov [slot],t1_2					

Après itération:



Coloriage itératif

- Quand des noeuds sont marqués, on itére:
  - On produit le code avec le vidage (ici t1)
  - On recommence l'allocation
- Terminaison: à chaque itération, la pression registre diminue
- Complétude: Au pire, tous les registres seront vidés => pression = 2 registres <= K

Coloriage itératif, algorithme

Allouer(G,K)

- Colorier(G,K) --> alloc
- **Tant qu'il existe des noeuds marqués**
  - Vidage := Vidage U noeuds marqués
  - Ajouter le code de vidage
  - Produire le nouveau graphe d'interférence, G'
  - Colorier(G') --> alloc
- Colorier(G | Vidage,+oo) --> alloc_pile
- Retourner alloc U alloc_pile

Précoloriage

La traduction dirigée par la syntaxe et la sélection forcent l'allocation de certains temporaires:

- call/return --> rax
- mul/div --> rdx et rax
- Il suffit de précolorier le graphe avant l'allocation!